

# **ENHANCED BOOLEAN PROCESSOR**

## **CROSS-REFERENCE TO RELATED APPLICATION**

**[0001]** This application is a continuation-in-part of U.S. Patent Application Serial No. 10/075,917 filed February 13, 2003, which claims the benefit of provisional U.S. Patent Application Serial No. 60/268,471 filed February 14, 2001; provisional U.S. Patent Application Serial No. 60/268,472 filed February 14, 2001; and provisional U.S. Patent Application Serial No. 60/268,478 filed February 14, 2001.

**[0002]** In addition, this application is entitled to the benefit of, and claims priority to, provisional U.S. Patent Application Serial No. 60/454,684 filed March 17, 2003 and entitled “BOOLEAN CO-PROCESSOR;” provisional U.S. Patent Application Serial No. 60/479,586 filed June 19, 2003 and entitled “BOOLEAN PROCESSING MEMORY & DEVICE SYNCHRONIZATION;” provisional U.S. Patent Application Serial No. 60/483,061 filed June 30, 2003 and entitled “BOOLEAN PROCESSOR EXPANDED SHORT-CIRCUIT ADDRESSING;” provisional U.S. Patent Application Serial No. 60/495,822 filed August 18, 2003 and entitled “BOOLEAN PROCESSOR INTER-DEVICE COMMUNICATIONS;” provisional U.S. Patent Application Serial No. 60/520,395 filed November 17, 2003 and entitled “BOOLEAN PROCESSOR PERFORMANCE ENHANCEMENTS;” and provisional U.S. Patent Application Serial No. 60/537,350 filed January 20, 2004 and entitled “BOOLEAN ACCELERATOR TECHNOLOGY;” the entirety of each of which is incorporated herein by reference.

## **BACKGROUND OF THE PRESENT INVENTION**

### **Field of the Present Invention**

**[0003]** The present invention relates generally to the computing and microelectronics fields. More specifically, the present invention relates to a Boolean-based processor architecture that is capable of the short-circuit evaluation of Conjunctive Normal Form (“CNF”) Boolean expressions, Disjunctive Normal Form (“DNF”) Boolean expressions, or both. The Boolean processor of the present invention provides an inexpensive, scalable, and efficient means for

computing in environments typically suited for application-specific microprocessors and microcontrollers, such as monitoring and automation environments.

### Background

**[0004]** A microprocessor is a general-purpose computing architecture, also known as a central processing unit (CPU). The microprocessor includes an arithmetic logic unit (ALU), an accumulator, a plurality of registers, a program counter, a stack pointer, a clock circuit, and a plurality of interrupt circuits. In building a complete computing system, the microprocessor must be supplemented with external components, such as a random-access memory (RAM) and a read-only memory (ROM), an oscillator, a plurality of memory decoders, a plurality of Input/Output (I/O) interfaces (ports), and a plurality of other devices, such as video displays and disk drives. The microprocessor is designed to perform a wide variety of calculations with data and return the results to a user or another machine. The microprocessor achieves this computing power through the use of a sophisticated instruction set that may contain a plurality of instructions for performing arithmetic operations, bit movement operations, memory fetch and store operations, etc. Because of the complexity of the calculations that the microprocessor performs, the programs that control its operation are generally relatively large, requiring the use of mass storage devices to house them. When needed for a specific calculation or task, a program is loaded into the system RAM and executed by the microprocessor.

**[0005]** The primary design factors related to the microprocessor are flexibility and expandability, allowing the microprocessor to handle almost any task. This adaptability has resulted in a relatively large demand for the microprocessor and has enabled manufacturers to mass-produce them, resulting in a relatively inexpensive and disposable product.

**[0006]** Like the microprocessor, a microcontroller is also a general-purpose computing architecture. The microcontroller differs from the microprocessor, however, in that it can operate as a complete, stand-alone computer system. The microcontroller includes all of the components of the microprocessor, in addition to its own RAM, ROM, plurality of counters, and I/O ports. The microcontroller is also relatively flexible and can be used in a plurality of applications, however, the microcontroller is intended for use in a relatively static environment,

requiring its programs to change minimally over time. The microcontroller is primarily intended to be used to control the environment within which it operates. The microcontroller is typically used in embedded system applications for monitoring and automation purposes. The microcontroller can be found in, for example, appliances (such as microwave ovens, refrigerators, televisions, VCRs, and stereos), automobiles (such as in engine control systems, diagnostics systems, and climate control systems), environmental control systems (such as in factories, greenhouses, and homes), instrumentation arrays, and aerospace systems.

**[0007]** The microprocessor differs from the microcontroller in their sets of operational codes. The microprocessor has far more operational codes for moving data to and from an external memory than the microcontroller, which may only have a few such operational codes. From an internal bit-handling perspective, the microcontroller has far more internal bit-handling operational codes than the microprocessor, which may only have a few. The architecture of both the microprocessor and the microcontroller are intended for mass use and are designed for flexibility and expandability. Each has the goal of supporting a wide range of applications. While the primary use of the microprocessor is for calculation-intensive computing, the microcontroller is designed to handle smaller calculations and to control its environment.

**[0008]** The short-circuit evaluation of a Boolean expression or operation is simply the abandonment of the remainder of the expression or operation once its value has been determined. If the outcome of the expression or operation can be determined prior to its full evaluation, it makes sense to save processing cycles by avoiding the remaining, unnecessary, conditional tests of the expression or operation. In other words, the short-circuit evaluation of a Boolean expression is a technique that specifies the partial evaluation of expressions involving AND and OR operations.

**[0009]** What is needed is a microprocessor and/or a microcontroller that is capable of evaluating complex Boolean expressions that are in Conjunctive Normal Form (CNF). Disjunctive Normal Form (DNF) Boolean expressions can also be incorporated into the

architecture of the microprocessor and/or the microcontroller, however, there are inefficiencies associated with the processing of the DNF equivalents of CNF expressions.

**[0010]** A Boolean expression is in DNF if it is expressed as the sum (OR) of products (AND). That is, the Boolean expression B is in DNF if it is written as:

$$A_1 \text{ OR } A_2 \text{ OR } A_3 \text{ OR } \dots \text{ } A_n$$

where each term  $A_i$  is expressed as:

$$T_1 \text{ AND } T_2 \text{ AND } \dots \text{ AND } T_m$$

where each term  $T_i$  is either a simple variable, or the negation (NOT) of a simple variable. Each term  $A_i$  is referred to as a “minterm”. A Boolean expression is in CNF if it is expressed as the product (AND) of sums (OR). That is, the Boolean expression B is in CNF if it is written as:

$$O_1 \text{ AND } O_2 \text{ AND } O_3 \text{ AND } \dots \text{ } O_n$$

where each term  $O_i$  is expressed as:

$$T_1 \text{ OR } T_2 \text{ OR } \dots \text{ OR } T_m$$

where each term  $T_i$  is either a simple variable, or the negation (NOT) of a simple variable. Each term  $O_i$  is referred to as a “maxterm”. The terms “minterm” and “maxterm” can also be referred to as “disjunct” and “conjunct”, respectively.

**[0011]** The short-circuit evaluations of a CNF Boolean expression and a DNF Boolean expression are handled differently. In the case of a CNF expression, short-circuiting can occur if any of the conjuncts evaluates to false. In the following example,

$$(A \vee B) \wedge (C \vee D)$$

if either of the conjuncts,  $(A \vee B)$  or  $(C \vee D)$ , evaluates to false, the expression also evaluates to false. If  $(A \vee B)$  evaluates to false, the remainder of the expression can be eliminated, thereby saving the time required to evaluate the other conjunct. In contrast to CNF short-circuit

evaluation, a DNF expression can be short-circuited if any of the disjuncts evaluates to true. Using the previous example in DNF,

$$(A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D)$$

if any of the disjuncts,  $(A \wedge C)$ ,  $(A \wedge D)$ ,  $(B \wedge C)$ , or  $(B \wedge D)$ , evaluates to true, the expression also evaluates to true. For example, if  $(A \wedge C)$  evaluates to true, the evaluation of the remaining three disjuncts can be eliminated, since their values are irrelevant to the outcome of the expression.

**[0012]** Thus, the short-circuit evaluation of both CNF and DNF expressions becomes increasingly valuable, in terms of cycle savings, as the complexity of the expressions increases. In large scale monitoring and automation applications, the short-circuit evaluation of both CNF and DNF expressions is essential.

**[0013]** The Boolean processor is an architecture that is designed to provide optimal performance for computing complex Boolean expressions. The Boolean processor is intended for use in, among other things, monitoring and automation applications. The Boolean processor is built for speed and efficiency via its ability to perform the short-circuit evaluation of Conjunctive Normal Form (CNF) Boolean expressions. It provides enhanced computing performance, in terms of the number of instructions required to perform equivalent operations, to that of other general-purpose architectures. The Boolean processor is described in USPTO Patent Application Number 10/075,917. This processor will be referred to as the CNF Boolean processor.

**[0014]** A Disjunctive Normal Form (DNF) Boolean processor is also described in the Provisional Patent Application titled “DNF Boolean Processor” by Kenneth E. Koch III. The DNF Boolean processor is a similar architecture to the Boolean processor with the exception that it is designed to perform short-circuit evaluations on Boolean expressions in Disjunctive Normal Form.

## SUMMARY OF THE PRESENT INVENTION

**[0015]** Embodiments of the general-purpose Boolean processor of the present invention incorporate an architecture that is designed to provide optimal performance for computing complex Boolean expressions. The Boolean processor is intended for use in, among other things, monitoring and automation applications. The Boolean processor is built for speed and efficiency via its ability to perform the short-circuit evaluation of Conjunctive Normal Form (CNF) Boolean expressions. The Boolean processor provides enhanced computing performance, in terms of the number of instructions required to perform equivalent operations, to that of other general-purpose architectures.

**[0016]** In one embodiment of the present invention, a processor includes a Boolean logic unit, wherein the Boolean logic unit is operable for performing the short-circuit evaluation of Conjunctive Normal Form Boolean expressions/operations, a plurality of input/output interfaces, wherein the plurality of input/output interfaces are operable for receiving a plurality of compiled Boolean expressions/operations and transmitting a plurality of compiled results, and a plurality of registers.

**[0017]** In another embodiment of the present invention, a processing method includes starting an operation related to a Conjunctive Normal Form Boolean expression, wherein the Boolean expression comprises a conjunct, evaluating the conjunct, and selectively short-circuiting a portion of the Boolean expression.

**[0018]** In a further embodiment of the present invention, a device polling unit for finding new devices, assigning addresses to those devices, polling those devices for their current states, and updating a random-access memory with those states includes a maximum device address electrically-erasable programmable read-only memory, wherein the electrically-erasable programmable read-only memory is operable for storing the highest address of all known devices on a system, wherein the electrically-erasable programmable read-only memory includes an increment line that increments its value by one whenever it is asserted and a plurality of output lines that continuously output its value. The device polling unit also includes an *n*-bit

incrementing register, wherein the  $n$ -bit incrementing register is operable for holding an  $n$ -bit number representing a current address of a device being polled, wherein the  $n$ -bit incrementing register includes a reset line that sets the register to ‘zero’ whenever it is asserted, and wherein the  $n$ -bit incrementing register further includes an increment line and a plurality of output lines that continuously output its value to an AND unit and a current address encoder. The device polling unit operates in a continuous loop after it is started.

**[0019]** In a further embodiment of the present invention, a device interface unit for listening for new device seek, new address, state enable, and control line assertions and determining whether or not there is work to do as a result of such assertions includes a new device electrically-erasable programmable read-only memory, wherein the new device electrically-erasable programmable read-only memory includes an  $n$ -bit store that is initially set to ‘one’, and wherein, when a new device seek line is asserted, the  $n$ -bit store asserts a new device found line. The device interface unit also includes an address decoder, wherein, if the  $n$ -bit store is set, it allows an address passed on a new address line to be placed in an  $n$ -bit address electrically-erasable programmable read-only memory and the  $n$ -bit store to be cleared. The device interface unit further includes a control word decoder, wherein the control word decoder is operable for reading serial bits off of a control line, and wherein, if an address matches the address in the  $n$ -bit address electrically-erasable programmable read-only memory, a plurality of control bits output to a device controller to change its state. The device interface unit further includes an address and state encoder, wherein the address and state encoder is operable for reading bits in parallel that represent the address and state of the device and serially outputs the bits to a receiver.

**[0020]** One new invention is a Boolean co-processor that will be incorporated into larger chip designs such as other microcontrollers and/or microprocessors. The co-processor will consist of a Boolean processor and/or DNF Boolean processor that is/are modified to accept portions of code from the host unit (microcontroller and/or microprocessor). These portions of code represent Boolean expressions in Conjunctive Normal Form (CNF) and/or Disjunctive Normal Form (DNF) and are off-loaded to the Boolean co-processor to maximize the overall speed of the host unit as shown in Fig. 20. In addition to providing the results of the evaluation

of Boolean expressions to the host architecture, the Boolean co-processor may also interface directly with devices connected to the entire system as shown in Fig. 21.

**[0021]** This functionality serves a similar purpose to that of a math co-processor with the exception that it executes Boolean operations instead of mathematical operations.

**[0022]** The efficiency of the short circuiting of CNF expressions can be maximized by:

C1. Evaluating terms within conjuncts that are most likely to be true as early as possible in the overall evaluation of each conjunct.

C2. Evaluating conjuncts that are most likely to evaluate to false as early as possible in the overall evaluation of the CNF expression.

**[0023]** Another invention is a method by which the probabilities of terms within conjuncts evaluating to true and/or false and the probabilities of conjuncts evaluating to true and/or false are stored in any form or fashion and recorded as the CNF expressions are evaluated over multiple iterations. These results are then used to recompile and/or reconfigure the ordering of terms, conjuncts, and/or operations to maximize the efficiency of the evaluations as described above in C1 and C2. The flow chart in Fig. 19 shows the process by which the efficiency of the evaluation of both CNF and DNF expressions is maximized.

**[0024]** The efficiency of the short circuiting of DNF expressions can be maximized by:

D1. Evaluating terms within disjuncts that are most likely to be false as early as possible in the overall evaluation of each disjunct.

D2. Evaluating disjuncts that are most likely to evaluate to true as early as possible in the overall evaluation of the DNF expression.

**[0025]** Another invention is a method by which the probabilities of terms within disjuncts evaluating to true and/or false and the probabilities of disjuncts evaluating to true and/or false are stored in any form or fashion and recorded as the DNF expressions are evaluated over multiple iterations. These results are then used to recompile and/or reconfigure the ordering of terms, disjuncts, and/or operations to maximize the efficiency of the evaluations as described above in D1 and D2.

**[0026]** Another invention is an architecture similar to that previously disclosed that performs short-circuit evaluation of Disjunctive Normal Form Boolean expressions. In the case of DNF, an AND register is used to evaluate the results of and perform short circuiting within disjuncts when a state returns a false value, and roll the results into an OR register that will perform a short circuit operation if any of the disjuncts in the expression being evaluated results in a true value.

**[0027]** The commonly-assigned of U.S. Patent Application Serial No. 10/075,917 describes an AND register that is initialized to a value of true ('one'), remains at true ('one') if all conjuncts of a Boolean expression/operation being evaluated are true, and changes to a state of false ('zero') if the expression being evaluated is false. In another invention, the flexibility of the AND register is further extended. The AND register may be modified such that one or more values may be used to initialize the register and represent a true ('one') value. The same applies to a false ('zero') value, where any of another set of values, other than those used to represent a true ('one') value, may be used to represent a false ('zero') value.

**[0028]** The commonly-assigned of U.S. Patent Application Serial No. 10/075,917 also describes an OR register that is initialized to a value of 'zero' and remains at 'zero' until a state in a conjunct evaluates to 'one', at which point the register is set to 'one' and the operation is short-circuited to the start of the next conjunct in the expression being evaluated. In another invention, the flexibility of the OR register is extended. The OR register may be modified such that one or more values may be used to initialize the register and represent a false ('zero') value. The same applies to a true ('one') value, where any of another set of values, other than those used to represent a false value, may be used to represent a true value.

**[0029]** The commonly-assigned of U.S. Patent Application Serial No. 10/075,917 also describes an instruction register comprised of  $n+m+3$  bits consisting of an  $n$ -bit address, an  $m$ -bit control/state word, and a 3-bit operational code. In this case,  $n$  and  $m$  can be any number. In another invention, the flexibility of the instruction register is further extended by making it an  $n+m+x$  bit register where  $x$  represents an operational code of  $x$  bits, where  $x$  can be any number. This will permit subsequent designs with more than  $2^3$  operational codes (or less than  $2^3$  operational codes, if desired).

**[0030]** In another invention, Boolean processor device states are stored in the Device State Storage portion, or Random Access Memory, of the architecture. To ensure system accuracy, it is important that a memory location is not read while its contents are being modified. Doing so could result in erroneous results. To ensure that the aforementioned situation does not occur, the Boolean processor architecture may be modified so that the processor either waits for the modification operation to end before reading a location in memory, or skips the operation. In the event that the value of the memory location is critical to the operation being performed by the system, putting the system in a wait state is preferable.

**[0031]** The addition of a wait state or skip operation can be achieved by adding some form of indicator, including, but not limited to, a single bit added to each memory location, that will indicate whether or not a memory location is in the process of being modified. The processor will then wait for the modification to end before accessing the location or skip the reading of the location.

**[0032]** Additionally, the processor writes state change information directly across a bus to devices attached to it. The processor is designed to process its micro-program at a much faster rate than its devices operate. In the event that two or more device states are changed in a timeframe that is shorter than the time required to update a device's state, a buffer, as shown in Fig. 35, will need to be used to store state change commands until the bus is clear. Once an update message has been sent, the next message in the buffer may be sent. To ensure system accuracy, the single bit in RAM that indicates that the state of the corresponding device is being

modified can be set to true to prevent the system from reading an expired state from device state storage.

**[0033]** Another method for updating device states would be the addition of another RAM module that will store updated states. The Boolean processor would write state updates to that module. The devices could then request their own updated states from the RAM module. This configuration would operate similarly to the configuration described above except that, instead of having device states “pushed” out to the devices via the control encoder, devices would “pull” their states from the additional RAM module, as shown in Fig. 36.

**[0034]** In another invention, multiple instances of the Boolean processor may be used in parallel to evaluate complex CNF or Disjunctive Normal Form (DNF) expressions in a divide-and-conquer type mode. In the case of CNF, the expression’s conjuncts would be distributed to the different processors for evaluation. In the event that a conjunct, or series of conjuncts, results in a false evaluation, the processor upon which the conjunct(s) were being evaluated will signal the other processors that the entire operation is false whereby causing the next Boolean expression to be distributed among the processors for evaluation.

**[0035]** In the case of DNF, the expression’s disjuncts would be distributed to the different processors for evaluation. In the event that a disjunct, or series of disjuncts, results in a true evaluation, the processor upon which the disjunct(s) were being evaluated will signal the other processors that the entire operation is true whereby causing the next expression to be distributed among the processors for evaluation.

**[0036]** Instances of both DNF and CNF Boolean processors may also be intermingled to handle expressions. For example, in the case of the following CNF expression:

If (A or B or C or D) and (E or F or G) and H then I

where A, B, C, D, E, F, G, and H are terms of the form  $x = y$  where x represents a device state and y represents a value for comparison,

two DNF Boolean processors could be employed to evaluate the first two conjuncts since each conjunct represents a DNF expression in its most simple form. The final values from the DNF Boolean processors as well as the evaluation of H could then be rolled into a CNF Boolean processor as shown in Fig. 43.

**[0037]** The original application specified that both the next operation address register and the end of OR address register are loaded with values specified in the address portion of the instruction register. These values specify the addresses of lines of code within the micro-program that are jumped to when performing short circuit operations. This design limits the number of micro-program lines (or micro-program addresses) that can be accessed by the next operation address register and the end of OR address register to  $2^n$ , where n is the width, in bits, of the address portion of the instruction register.

**[0038]** In another invention, in order to expand the micro-program address values that can be stored in the next operation address register and the end of OR address register, the architecture will be modified to use the bits from both the address and control/state portions of the instruction register when loading the next operation address and the end of OR address registers with the values of micro-program addresses.

**[0039]** Another solution is to modify the control store portion of the architecture to include discrete jump to addresses that would only be utilized on instructions that are capable of being jumped to. While the limit on the number of instructions that may be jumped to remains the same in this case, the discrete jump addresses permits the “jump to” addresses to be dispersed throughout the entire micro-program, as opposed to being limited to the first  $2^n$  instructions, where n is the width, in bits, of the address portion of the instruction register.. The previously mentioned solution, however, in which the address and control/state portions of the instruction register are utilized, is the preferred solution.

**[0040]** Another invention is a modification to the Boolean processor that will enable inter-device communication. This functionality will permit the state of any device

connected to the Boolean processor to be sent directly to any other device in the system. This communication may be achieved in one of two ways:

1. Via the addition of an extra instruction that will accept the addresses of the two devices (between which state information will be transferred) and facilitate the exchange of state data, as shown in Fig. 41. This change will require the expansion of the operation code (OP code) portion of the Boolean processor's instruction register by one bit to accommodate the extra instruction.
2. Via the addition of an indicator bit to the instruction register that will indicate that the value in the Control/State portion of the instruction register is either a direct value or the address of another device, as shown in Fig. 42. In the latter case, the state of the source device will be sent to the target device.

**[0041]** This new invention may be used for, but not limited to, applications such as serial and parallel communications add-ons for existing architectures. In either application (serial or parallel communications), peripherals (printers, modems, pointing devices, etc.), as well as internal memory locations, can be represented as devices within the Boolean processor architecture.

**[0042]** In the event that the Boolean processor is used as a serial communications co-processor in an existing microcontroller or microprocessor architecture, registers in the host architecture may be accessed directly by the Boolean processor by representing them as devices.

**[0043]** Receiving and transmitting acknowledgements can be represented as discrete states for each device (or memory location in the host architecture).

**[0044]** Another invention is a modification to the Boolean processor that will enhance the performance of the architecture via the inclusion of:

1. An arithmetic logic unit (ALU). The ALU may be as simple as, but not limited to, a two word comparator, where the word size is determined by the number of device states supported by the implementation of the architecture. A two word

comparator would simply test one word to determine if its value was either greater than, less than, greater than or equal to, or less than or equal to, the value of the other word. This will allow the Boolean processor to test threshold values and eliminate the need to test every discrete value in a range when making threshold comparisons (ex: x is greater than y).

In the case of the ALU being a two word comparator, four new instructions would need to be built into the architecture. These instructions would support an AND compare between an immediate value and a device state, an AND compare between two device states, an OR compare between an immediate value and a device state, and an OR compare between two device states.

2. Two new instructions to enable AND and OR operations when comparing two device state values for equality. The current AND and OR operations are used to compare immediate values with a single device state.
3. An additional instruction for loading a value directly into an address in device state storage. This instruction will minimize the number of instructions required to load a value into a device state storage address.

**[0045]** The enhancements mentioned above will improve the performance of the Boolean processor in stand-alone applications as well as co-processor applications. The addition of these enhancements enables the Boolean processor to be used in applications such as data mining, knowledge discovery, and artificial intelligence where complex Boolean Normal Form (Conjunctive and Disjunctive) evaluations are commonplace.

**[0046]** Additional inventions are a set of Boolean processor cores. The cores are designed to rapidly compute the results of CNF and/or DNF expressions by interfacing the n-bit AND, OR, OR conjunct, and AND disjunct registers of the CNF and DNF Boolean processors with any existing computing architecture. These cores may also utilize the instruction set of the host architecture and directly accept any value from the host architecture, such as program addresses and the host's condition bit, as input.

**[0047]** The CNF Boolean processor core is comprised of an AND register, an OR register, and an OR conjunct register. These registers are used to compute the outcome of CNF Boolean expressions and their functionality in computing CNF Boolean expressions is described in USPTO Patent Application Number 10/075,917. In addition to these three registers, a next operation address register, which holds the address of the instruction immediately following the CNF expression, as well as an end of OR address register, which holds the address of the instruction immediately following an OR conjunct, may also be included in the core. These address registers may also be omitted and substituted with registers from the host architecture.

**[0048]** The AND, OR, and OR conjunct registers may be set and/or reset in the following ways:

- 1) By modifying their values directly with load instructions (from the host architecture and/or the CNF Boolean processor core) or any other register modifying instruction (from the host architecture and/or the CNF Boolean processor core).
- 2) By utilizing another register (within the host architecture or the CNF Boolean processor core) that can be set with a single instruction or series of instructions and will, subsequently, modify the values of the AND, OR, and OR conjunct registers by tying bits of the register directly to them.
- 3) In the event that address registers are used with the CNF Boolean processor core, modification of the address registers can trigger the setting and resetting of the AND, OR, and OR conjunct registers.

**[0049]** DNF Boolean processor core is comprised of an AND register, an OR register, and an AND disjunct register. These registers are used to compute the outcome of DNF Boolean expressions and their functionality in computing DNF Boolean expressions is described in the Provisional Patent Application titled “DNF Boolean Processor” by Kenneth E. Koch III. In addition to these three registers, an end of operation address register, which holds the address of the instruction that is executed in the event that the DNF expression evaluates to true, as well

as an end of AND address register, which holds the address of the instruction immediately following an AND disjunct, may also be included in the core. These address registers may also be omitted and substituted with registers from the host architecture.

**[0050]** The AND, OR, and AND disjunct registers may be set and/or reset in the following ways:

- 1) By modifying their values directly with load instructions (from the host architecture and/or the DNF Boolean processor core) or any other register modifying instruction (from the host architecture and/or the DNF Boolean processor core).
- 2) By utilizing another register (within the host architecture or the DNF Boolean processor core) that can be set with a single instruction or series of instructions and will, subsequently, modify the values of the AND, OR, and AND disjunct registers by tying bits of the register directly to them.
- 3) In the event that address registers are used with the DNF Boolean processor core, modification of the address registers can trigger the setting and resetting of the AND, OR, and AND disjunct registers.

**[0051]** The functionality of both the CNF Boolean processor core and the DNF Boolean processor core may be combined to form a single core that is capable of computing the results of both CNF and DNF Boolean expressions.

**[0052]** The input to a Boolean processor core from a host architecture may vary and may include, but is not limited to:

- 1) An indication to the core that the host architecture is in need of its services. This may include a register that, when set, will output the values of the address registers in the event that any Boolean short-circuiting occurs.

- 2) Addresses for the next operation address register, addresses for the end of operation address register, addresses for the end of OR address register (CNF) and addresses for the end of AND address register (DNF).
- 3) Condition bit or similar true/false comparison indication from the host architecture.
- 4) Direct values for the AND, OR, OR conjunct (CNF), and AND disjunct (DNF) registers.
- 5) A word, register, memory value, or any portion of the aforementioned as a direct feed into any of the Boolean processor core registers.
- 6) Any combination of the above.

**[0053]** The output from a Boolean processor core to its host architecture may include, but is not limited to:

- 1) A direct update of the host's program counter to the instruction address specified in the next operation address register (CNF), the end of operation address register (DNF), the end of OR address register (CNF), or the end of AND address register (DNF).
- 2) The execution of an instruction by the host architecture that will make the value of any of a Boolean processor core's register values accessible to the host architecture.
- 3) A feed to an interrupt in the host architecture triggered by the changing of any of a Boolean processor's register values.

**[0054]** Further areas of applicability of the present invention will become apparent from the detailed description provided hereinafter. It should be understood that the detailed description and specific examples, while indicating the preferred embodiment of the invention,

are intended for purposes of illustration only and are not intended to limit the scope of the invention.

#### **BRIEF DESCRIPTION OF THE DRAWINGS**

**[0055]** Further features, embodiments, and advantages of the present invention will become apparent from the following detailed description with reference to the drawings, wherein:

**[0056]** Fig. 1 is a schematic block diagram of a conventional microprocessor;

**[0057]** Fig. 2 is a schematic block diagram of a conventional microcontroller;

**[0058]** Fig. 3 is a schematic block diagram of one embodiment of the Boolean processor of the present invention;

**[0059]** Fig. 4 is a schematic diagram of a CNF embodiment of the architecture of the Boolean processor of the present invention;

**[0060]** Fig. 5 is a schematic diagram of a DNF embodiment of the architecture of the Boolean processor of the present invention;

**[0061]** Fig. 6 is a graph of a DNF/CNF ratio using inter-term short-circuit evaluation, with 1 control state and a small number of “other” states;

**[0062]** Fig. 7 is a graph of a DNF/CNF ratio using inter-term short-circuit evaluation, with 1 control state and a large number of “other” states;

**[0063]** Fig. 8 is a graph of a DNF/CNF ratio using inter-term short-circuit evaluation, with 5 control states and a small number of “other” states;

**[0064]** Fig. 9 is a graph of a DNF/CNF ratio using inter-term short-circuit evaluation, with 5 control states and a large number of “other” states;

**[0065]** Fig. 10 is a graph of a DNF/CNF ratio using inter-term short-circuit evaluation, with 10 control states and a small number of “other” states;

**[0066]** Fig. 11 is a graph of a DNF/CNF ratio using inter-term short-circuit evaluation, with 10 control states and a large number of “other” states;

**[0067]** Fig. 12 is a graph of a DNF/CNF ratio using inter/intra-term short-circuit evaluation, with 1 control state and a small number of “other” states;

**[0068]** Fig. 13 is a graph of a DNF/CNF ratio using inter/intra-term short-circuit evaluation, with 1 control state and a large number of “other” states;

**[0069]** Fig. 14 is a graph of a DNF/CNF ratio using inter/intra-term short-circuit evaluation, with 10 control states and a small number of “other” states;

**[0070]** Fig. 15 is a graph of a DNF/CNF ratio using inter/intra-term short-circuit evaluation, with 10 control states and a large number of “other” states;

**[0071]** Fig. 16 is a graph of the relative performance of the Intel 8051 microcontroller versus the Boolean processor of the present invention;

**[0072]** Fig. 17 is a graph of the relative performance of the Intel 8086 family of microprocessors versus the Boolean processor of the present invention;

**[0073]** Fig. 18 is a graph of the relative performance of the Motorola MMC2107 microcontroller versus the Boolean processor of the present invention;

**[0074]** Fig. 19 is a flowchart diagram of a re-compiling process for use with the preferred embodiments of the present invention;

**[0075]** Fig. 20 is a high-level block diagram illustrating the functional relationship between a host computer device and a Boolean co-processor;

**[0076]** Fig. 21 is a high-level block diagram illustrating the functional relationship between a host computer device and a Boolean co-processor having a direct interface with one or more system devices;

**[0077]** Fig. 22 is a schematic diagram of a CNF embodiment of a Boolean processor core in accordance with another preferred embodiment of the present invention;

**[0078]** Fig. 23 is a detailed schematic diagram of a circuit suitable for use as either of the address registers shown in Fig. 22;

**[0079]** Fig. 24 is a detailed schematic diagram of a circuit suitable for use as any of the 1-bit registers shown in Fig. 22;

**[0080]** Fig. 25 is a detailed schematic diagram of the 4-input AND gate circuit 441 of Fig. 22;

**[0081]** Fig. 26 is a detailed schematic diagram of a circuit suitable for use as either of the address output AND gates shown in Fig. 22;

**[0082]** Fig. 27 is a detailed schematic diagram of the address OR output circuit;

**[0083]** Fig. 28 is a schematic diagram of a DNF embodiment of a Boolean processor core in accordance with another preferred embodiment of the present invention;

**[0084]** Fig. 29 is a detailed schematic diagram of the 3-input AND gate circuit of Fig. 28;

**[0085]** Fig. 30 is a schematic diagram of a combined CNF/DNF Boolean processor core in accordance with another preferred embodiment of the present invention;

**[0086]** Fig. 31 is a schematic diagram of one embodiment of the architecture of the device polling unit of the present invention;

**[0087]** Fig. 32 is a schematic diagram of one embodiment of the architecture of the device interface unit of the present invention;

**[0088]** Fig. 33 is a schematic diagram of one embodiment of a system that includes the Boolean processor and the device polling unit of the present invention;

**[0089]** Fig. 34 is a schematic diagram of one embodiment of a system that includes  $n$  devices, each incorporating the device interface unit of the present invention;

**[0090]** Fig. 35 is a high-level block diagram illustrating a first approach for ensuring data integrity in a Boolean processor system;

**[0091]** Fig. 36 is a high-level block diagram illustrating a second approach for ensuring data integrity in a Boolean processor system;

**[0092]** Fig. 37 is a schematic diagram of a CNF Boolean processor having a first enhanced logic unit in accordance with another preferred embodiment of the present invention;

**[0093]** Fig. 38 is an enlarged schematic diagram of a portion of the Boolean processor of Fig. 37;

**[0094]** Fig. 39 is a schematic diagram of a CNF Boolean processor having a second enhanced logic unit in accordance with another preferred embodiment of the present invention;

**[0095]** Fig. 40 is an enlarged schematic diagram of a portion of the Boolean processor of Fig. 39;

**[0096]** Fig. 41 is an enlarged schematic diagram of a portion of a Boolean processor having a direct connection for facilitating inter-device communication via the device state storage in accordance with another preferred embodiment of the present invention;

**[0097]** Fig. 42 is an enlarged schematic diagram of a portion of a Boolean processor having a direct connection for facilitating inter-device communication via the device state storage in accordance with yet another preferred embodiment of the present invention; and

**[0098]** Fig. 43 is a high-level block diagram illustrating the use of Boolean processors in a parallel processing environment.

#### **DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS**

**[0099]** Referring now to the drawings, in which like numerals represent like components throughout the several views, the preferred embodiments of the present invention are next described. The following description of the preferred embodiment(s) is merely exemplary in nature and is in no way intended to limit the invention, its application, or uses.

**[00100]** A microprocessor is a general-purpose computing architecture, also known as a central processing unit (CPU). Referring to Fig. 1, the microprocessor **10** includes an arithmetic logic unit (ALU) **12**, an accumulator **14**, a plurality of registers **16**, a program counter **18**, a stack pointer **20**, a clock circuit **22**, and a plurality of interrupt circuits **24**. In building a complete computing system, the microprocessor **10** must be supplemented with external components, such as a random-access memory (RAM) and a read-only memory (ROM), an oscillator, a plurality of memory decoders, a plurality of Input/Output (I/O) interfaces (ports), and a plurality of other devices, such as video displays and disk drives. The microprocessor **10**

is designed to perform a wide variety of calculations with data and return the results to a user or another machine. The microprocessor **10** achieves this computing power through the use of a sophisticated instruction set that may contain a plurality of instructions for performing arithmetic operations, bit movement operations, memory fetch and store operations, etc. Because of the complexity of the calculations that the microprocessor **10** performs, the programs that control its operation are generally relatively large, requiring the use of mass storage devices to house them. When needed for a specific calculation or task, a program is loaded into the system RAM and executed by the microprocessor **10**.

**[00101]** The primary design factors related to the microprocessor **10** are flexibility and expandability, allowing the microprocessor **10** to handle almost any task. This adaptability has resulted in a relatively large demand for the microprocessor **10** and has enabled manufacturers to mass-produce them, resulting in a relatively inexpensive and disposable product.

**[00102]** Like the microprocessor **10**, a microcontroller is also a general-purpose computing architecture. The microcontroller differs from the microprocessor **10**, however, in that it can operate as a complete, stand-alone computer system. Referring to Fig. 2, the microcontroller **26** includes all of the components of the microprocessor **10** (Fig. 1), in addition to its own RAM **28**, ROM **30**, plurality of counters **32**, and I/O ports **34**. The microcontroller **26** is also relatively flexible and can be used in a plurality of applications, however, the microcontroller **26** is intended for use in a relatively static environment, requiring its programs to change minimally over time. The microcontroller **26** is primarily intended to be used to control the environment within which it operates. The microcontroller **26** is typically used in embedded system applications for monitoring and automation purposes. The microcontroller **26** can be found in, for example, appliances (such as microwave ovens, refrigerators, televisions, VCRs, and stereos), automobiles (such as in engine control systems, diagnostics systems, and climate control systems), environmental control systems (such as in factories, greenhouses, and homes), instrumentation arrays, and aerospace systems.

**[00103]** The microprocessor **10** differs from the microcontroller **26** in their sets of operational codes. The microprocessor **10** has far more operational codes for moving data to and from an external memory than the microcontroller **26**, which may only have a few such operational codes. From an internal bit-handling perspective, the microcontroller **26** has far more internal bit-handling operational codes than the microprocessor **10**, which may only have a few. The architecture of both the microprocessor **10** and the microcontroller **26** are intended for mass use and are designed for flexibility and expandability. Each has the goal of supporting a wide range of applications. While the primary use of the microprocessor **10** is for calculation-intensive computing, the microcontroller **26** is designed to handle smaller calculations and to control its environment.

**[00104]** The short-circuit evaluation of a Boolean expression or operation is simply the abandonment of the remainder of the expression or operation once its value has been determined. If the outcome of the expression or operation can be determined prior to its full evaluation, it makes sense to save processing cycles by avoiding the remaining, unnecessary, conditional tests of the expression or operation. In other words, the short-circuit evaluation of a Boolean expression is a technique that specifies the partial evaluation of the expression involving an AND and/or an OR operation, or a plurality of each.

**[00105]** What is needed is a microprocessor and/or a microcontroller that is capable of evaluating complex Boolean expressions that are in Conjunctive Normal Form (CNF). Disjunctive Normal Form (DNF) Boolean expressions can also be incorporated into the architecture of the microprocessor and/or the microcontroller, however, there are inefficiencies associated with the processing of the DNF equivalents of CNF expressions.

**[00106]** A Boolean expression is in DNF if it is expressed as the sum (OR) of products (AND). That is, the Boolean expression B is in DNF if it is written as:

$$A_1 \text{ OR } A_2 \text{ OR } A_3 \text{ OR } \dots \text{ } A_n \quad (1)$$

where each term  $A_i$  is expressed as:

$$T_1 \text{ AND } T_2 \text{ AND } \dots \text{ AND } T_m \quad (2)$$

where each term  $T_i$  is either a simple variable, or the negation (NOT) of a simple variable. Each term  $A_i$  is referred to as a “minterm”. A Boolean expression is in CNF if it is expressed as the product (AND) of sums (OR). That is, the Boolean expression B is in CNF if it is written as:

$$O_1 \text{ AND } O_2 \text{ AND } O_3 \text{ AND } \dots \text{ AND } O_n \quad (3)$$

where each term  $O_i$  is expressed as:

$$T_1 \text{ OR } T_2 \text{ OR } \dots \text{ OR } T_m \quad (4)$$

where each term  $T_i$  is either a simple variable, or the negation (NOT) of a simple variable. Each term  $O_i$  is referred to as a “maxterm”. The terms “minterm” and “maxterm” can also be referred to as “disjunct” and “conjunct”, respectively.

**[00107]** The short-circuit evaluations of a CNF Boolean expression and a DNF Boolean expression are handled differently. In the case of a CNF expression, short-circuiting can occur if any of the conjuncts evaluates to false. In the following example,

$$(A \vee B) \wedge (C \vee D) \quad (5)$$

if either of the conjuncts,  $(A \vee B)$  or  $(C \vee D)$ , evaluates to false, the expression also evaluates to false. If  $(A \vee B)$  evaluates to false, the remainder of the expression can be eliminated, thereby saving the time required to evaluate the other conjunct. In contrast to CNF short-circuit evaluation, a DNF expression can be short-circuited if any of the disjuncts evaluates to true. Using the previous example in DNF,

$$(A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D) \quad (6)$$

if any of the disjuncts,  $(A \wedge C)$ ,  $(A \wedge D)$ ,  $(B \wedge C)$ , or  $(B \wedge D)$ , evaluates to true, the expression also evaluates to true. For example, if  $(A \wedge C)$  evaluates to true, the evaluation of the remaining three disjuncts can be eliminated, since their values are irrelevant to the outcome of the expression.

**[00108]** Thus, the short-circuit evaluation of both CNF and DNF expressions becomes increasingly valuable, in terms of cycle savings, as the complexity of the expressions increases. In large scale monitoring and automation applications, the short-circuit evaluation of both CNF and DNF expressions is essential.

**[00109]** Referring to Fig. 3, in one embodiment of the present invention, the architecture of the Boolean processor 36 can best be described as that of a microcontroller, at least functionally. The inputs of the microcontroller are compiled Boolean operations, or tests, and the outputs of the microcontroller are compiled result operations that are executed in conjunction with the results of the tests. The Boolean processor 36 includes a clock circuit 22, a program counter 18, a plurality of Input/Output (I/O) interfaces (ports) 34, a plurality of registers 16, a random-access memory (RAM) 28, and a read-only memory (ROM) 30. The Boolean processor 36 differs, however, from a conventional microcontroller in that the Boolean processor 36 does not contain an accumulator 14 (Figs. 1 and 2), a plurality of counters (other than the program counter 18), a plurality of interrupt circuits 24 (Figs. 1 and 2), or a stack pointer 20 (Figs. 1 and 2). Additionally, in lieu of an arithmetic logic unit (ALU) 12 (Figs. 1 and 2), the Boolean processor 36 includes a Boolean logic unit (BLU) 38. In terms of its size, speed, and functionality, the architecture of the Boolean processor 36 is designed to be inexpensive, scalable, and efficient. The Boolean processor 36 achieves these benefits through a simple design that is optimized for performing the short-circuit evaluation of complex Conjunctive Normal Form (CNF) Boolean expressions, Disjunctive Normal Form (DNF) Boolean expressions, or both.

**[00110]** The architecture of a CNF Boolean processor 36 is illustrated in Fig. 4. For the purposes of describing the architecture of the CNF Boolean processor 36, 8-bit device addressing and 8-bit control words are used. This results in the architecture of the CNF Boolean processor 36 supporting 256 devices, each device having 256 possible states. Optionally, the architecture of the CNF Boolean processor 36 can be scaled to accommodate  $2^n$  devices, each device having  $2^m$  possible states, where  $n$  and  $m$  are the number of device address bits and the number of possible states for each device, respectively. The defining feature of the architecture of the CNF Boolean processor 36 is its set of registers, or lack thereof. In contrast to

conventional microprocessors and microcontrollers, which can have a plurality of registers (typically from 8 to 64 bits wide), the CNF Boolean processor 36 has only six registers. Of the six registers, the instruction register 40, the next operation address register 42, and the end of OR address register 44 are the only registers which are generally required to be multi-bit registers. The remaining three registers 54, 56, 58 hold AND truth states, OR truth states, and an indicator for conjuncts containing OR clauses. Each of these registers 54, 56, 58 may be only a single bit in size, although additional bits may be included if desired.

**[00111]** The CNF Boolean processor 36 includes the instruction register 40, which is an  $n+m+x$ -bit wide register containing an  $n$ -bit address, an  $m$ -bit control/state word, and an  $x$ -bit operational code. Using 8-bit device addressing, 8-bit control words, and 3-bit operational codes, the instruction register 40 is 19 bits wide. The CNF Boolean processor 36 also includes a control store (ROM) 46, which is used to hold a compiled micro-program, including  $(n+m+x)$ -bit instructions. The CNF Boolean processor 36 further includes the program counter 18, which is used for fetching the next instruction from the control store 46. The CNF Boolean processor 36 further includes a memory (MUX) 48, which is used to configure the program counter 18 for normal operation, conditional jump operation, unconditional jump operation, and Boolean short-circuit operation. Six AND gates 50 and one OR gate 52 are used to pass operation results and a plurality of signals that are operational code dependent.

**[00112]** The AND register 54 is used to roll up the results of the conjuncts. If the AND register 54 is one bit in size, then the default value of the AND register 54 is one and it initializes to a value of one after a start of operational code. The 1-bit AND register 54 remains at a value of one if all of the conjuncts in the Boolean expression being evaluated are true. If this bit is set to zero at any time during the evaluation, the entire CNF operation is false. In such a case, the remainder of the operation may be short-circuited and the evaluation of the next operation can begin. It should be apparent, however, that the AND register 54 may be modified such that one or more alternative values may be used to initialize the register 54 and represent a “true” value. The same applies to a “false” value as well, where any of another set of values (provided that the selected value is different from the one(s) used to represent a “true” value) may be used to represent a “false” value.

**[00113]** The OR register **56** is used to roll up the results of each of the individual conjuncts. If the OR register **56** is one bit in size, then it initializes to a value of zero and remains in that state until a state in a conjunct evaluates to one. The OR conjunct register **58** is used to indicate that the evaluation of a conjunct containing OR clauses has begun. It initializes to a value of zero and remains in that state until an OR operation sets its value to zero. It should be apparent, however, that the OR register **56** may be modified such that one or more alternative values may be used to initialize the register **56** and represent a “false” value. The same applies to a “true” value as well, where any of another set of values (provided that the selected value is different from the one(s) used to represent a “false” value) may be used to represent a “true” value. Finally, if the OR conjunct register **58** is one bit in size, then it initializes to a value of zero and remains in that state until an OR operation sets its value to one. In the event that the 1-bit OR conjunct register **58** is set to one and the 1-bit OR register **56** is set to one, the entire conjunct evaluates to true and short-circuits to the start of the next conjunct.

**[00114]** The CNF Boolean processor **36** further includes an operation decoder **60**, which deciphers each operational code and controls the units that are dependent upon each operational code. In an embodiment preferred for its simplicity, the operational codes are 3 bits in length, and the functions of the operation decoder **60** by operational code include: Boolean AND (Op Code 0), Boolean OR (Op Code 1), End of Operation (Op Code 2), No Operation (Op Code 3), Unconditional Jump (Op Code 4), Conditional Jump (Op Code 5), Start of Operation (Op Code 6), and Start of Conjunct (Op Code 7). However, it will be apparent that the inclusion of one or more additional bits in the instruction register **40** would permit additional operational codes to be offered, and that the removal of a bit would reduce the number of operational codes offered, if either such design were to be desired.

**[00115]** A control encoder **62** accepts  $n+m$  bits in parallel (representing a device address and control word) and outputs them across a device bus (control lines) either serially or in parallel, depending upon the architecture of the given device bus. The next operation address register **42** stores the address used for Boolean short-circuiting. Short-circuiting occurs as soon as a conjunct evaluates to false. In such a case, the address is the address of the next operation. The end of OR address register **44** stores the address of the instruction immediately following a

conjunct containing OR clauses. It is used for the short-circuiting of conjuncts that contain OR clauses. The CNF Boolean processor **36** further includes a device state storage (RAM) **64**, which is responsible for storing the states of the devices that the CNF Boolean processor **36** monitors and/or controls. It has  $2^n$  addresses, each of which are  $m$ -bits wide, where  $n$  is the address width and  $m$  is the control/state word width, in bits.

**[00116]** The CNF Boolean processor **36** evaluates micro-programs and controls its environment based upon the results of the above-described evaluations. The micro-programs define the actions to be taken by devices in the event that given Boolean tests evaluate to true. The CNF Boolean processor **36** works on the principle that the devices will be controlled based upon their states and the states of other devices, or after some period of time has elapsed. In order to evaluate a micro-program, conditional tests must be compiled into CNF.

**[00117]** The CNF Boolean processor **36** performs eight functions, as specified by operational code. Op Code 0 – (Boolean AND) enables the AND gate **50** that loads the AND register **54** in the event that the conditional state of the device at the address in the instruction register **40** equals the state being tested in the instruction register **40**. The Boolean AND instruction serves two purposes. First, the instruction is used to rollup results between OR conjuncts. This is accomplished by comparing a “zero” value to the value in location 0, which always results in a “false” evaluation. Secondly, the instruction may be used to evaluate stand-alone conjuncts, in which case a value is being compared to a device state. Op Code 1 – (Boolean OR) sets the value of the OR conjunct register **58** to one, which enables short-circuiting within a conjunct containing OR clauses. Op Code 2 – (End of Operation) enables the AND gate **50** that AND’s the value of the OR register **56** with the value of the AND register **54**. If the AND register **54** evaluates to a value of one, the control encoder **62** is enabled and the address and control word specified in the end of operation code is sent to the proper device. Op Code 3 – (No Operation) does nothing. Op Code 4 – (Unconditional Jump) allows the MUX **48** to receive an address from an address portion of the instruction register **40** and causes an immediate jump to the instruction at that address. Op Code 5 – (Conditional Jump) provides that if the AND register **54** has a value of one, the test condition is met and the MUX **48** is enabled to receive the “jump to” address from the address portion of the instruction register **40**. Op Code 6 – (Start of

Operation) provides the address of the line following the end of operation line for the current operation. This address is used to short-circuit the expression and keep the CNF Boolean processor 36 from having to evaluate the entire CNF expression in the event that one of the conjuncts evaluates to zero. In addition to loading the next operation address into the next operation address register 42, this operation also sets the AND register 54 to one, the OR register 56 to zero and the OR conjunct register 58 to zero. Op Code 7 – (Start of OR Conjunct) provides the address of the line immediately following the conjunct and loads it into the end of OR address register 44. This address is used to provide short-circuiting out of a given conjunct in the event that one of the conjunct's terms evaluates to one.

**[00118]** The evaluation of a CNF expression begins with Start of Operation (Op Code 6) and proceeds to the evaluation of a conjunct. A conjunct may be either a stand-alone term (evaluated as an AND operation) or a conjunct containing OR clauses. In the latter case, each term of the conjunct is evaluated as part of an OR operation (Op Code 1). Each of these operations represents a test to determine if the state of a given device is equal to the state value specified in the corresponding AND or OR instruction. If the term evaluates to true, the OR-bit is set to a value of one. Otherwise, the OR-bit is set to a value of zero. In the case of a stand-alone term, this value automatically rolls up to the AND register 54. In conjuncts containing OR clauses, the result of each OR operation is OR'd with the current value of the OR register 56. This ensures that a true term anywhere in the conjunct produces a final value of true for the entire conjunct evaluation. In the event that the OR register 56 has a value of one and the OR conjunct register 58 is set to one, the conjunct will evaluate to true and may be short-circuited to the next conjunct. Next, the CNF Boolean processor 36 prepares for subsequent conjuncts (if any additional conjuncts exist). At this point, an AND operation (Op Code 0) joins the conjuncts and the value of the OR register 56 is rolled up to the AND register 54 by having the value of the OR register 56 AND'd with the value of the AND register 54. In the event that the OR-bit has a value of zero when the AND operation is processed, the AND-bit will change to a value of zero. Otherwise, the AND-bit's value will remain at one. If the AND-bit has a value of one, the next conjunct is evaluated. If the AND-bit has a value of zero, the final value of the CNF expression

is false, regardless of the evaluation of any additional conjuncts. At this point, the remainder of the expression may be short-circuited and the next CNF expression can be evaluated.

**[00119]** Preferably, the CNF Boolean processor 36 requires that functions be compiled in CNF. A micro-code compiler builds the micro-instructions such that they follow a CNF logic. The logic statements for CNF Boolean processor programs are nothing more than IF-THEN-ELSE statements. For example: IF (*Device A* has *State Ax*), THEN (Set *Device B* to *State By*), ELSE (Set *Device C* to *State Cz*). The logic of the IF expression must be compiled into CNF. The expression must also be expanded into a set of expressions AND'd together, and AND'd with a pre-set value of "true". For the CNF operation, the pre-set value of "true" is the initial value of the AND register 54 at the start of each logical IF operation. The above IF-THEN-ELSE statement would result in the following micro-code logic: [ (*Device A* has *State Ax*)  $\wedge$  "true" ]; if the AND statement is "true", then (SET *Device B* to *State By*); and if the AND statement is "false", then (SET *Device C* to *State Cz*).

**[00120]** The following are examples of how some common operations would be compiled to work with the architecture of the CNF Boolean processor 36. It should be noted that the Start of Operation Instruction (Op Code 6), as well as the Start of Conjunct Instruction (Op Code 7), have been omitted since ROM addresses are not listed in the examples. The notation in the following examples is of the form: DevX=Y, where X represents the device address and Y represents the current state of the device.

**[00121]      Example 1:**

*If (Dev1=8 or Dev2=0) and (Dev3=10 or Dev2=0) and (Dev4=1 or Dev2=0) then Dev9=20*

Micro-code		
Sequence #	Value of Instruction Register	Operation
1	00000001 00001000 001	OR
2	00000010 00000000 001	OR

3	00000000 00000000 000	AND
4	00000011 00001010 001	OR
5	00000010 00000000 001	OR
6	00000000 00000000 000	AND
7	00000100 00000001 001	OR
8	00000010 00000000 001	OR
9	00001001 00010100 010	End of Operation

---

**[00122]**      Example 2:

If Dev7=22 goto 200

*Else goto 100*

Micro-code		
Sequence #	Value of Instruction Register	Operation
1	00000111 00010110 001	OR
2	00000000 00000000 000	AND
3	00000000 11001000 101	Conditional Jump
4	00000000 01100100 100	Unconditional Jump

**[00123]**      As illustrated in Fig. 4, the next operation address register **42** and the end of OR address register **44** may be loaded with values from the  $n$ -bit “address” portion of the instruction register **40**. As described previously, these values specify the addresses of lines of code within the micro-program that are jumped to when performing short circuit operations. However, this design limits the number of micro-program lines (or micro-program addresses) that can be accessed by the next operation address register **42** and the end of OR address register **44** to  $2^n$ , where  $n$  is the width, in bits, of the address portion of the instruction register **40**.

**[00124]** In order to expand the micro-program address values that can be stored in the next operation address register **42** and the end of OR address register **44**, the architecture may be modified to use the bits from both the address and control/state portions of the instruction register **40** when loading the next operation address register **42** and the end of OR address register **44** with the values of micro-program addresses. This would expand the number of micro-program lines (or micro-program addresses) that can be accessed by the next operation address register **42** and the end of OR address register **44** to  $2^{n+m}$ , where  $n$  is the width, in bits, of the address portion of the instruction register **40** and  $m$  is the width, in bits, of the control/state portion of the instruction register **40**. This approach would require the “control/state” portion of the instruction register **40** to be connected directly to the address registers **42**, **44** in addition to the MUX **48**.

**[00125]** Another solution for expanding the range of micro-program address values that may be used is to modify the control store portion of the architecture to include discrete “jump to” addresses that would only be utilized on instructions that are capable of being jumped to. While the limit on the number of instructions that may be jumped to would remain the same in this case, the inclusion of discrete jump to addresses would permit the “jump to” addresses to be dispersed throughout the entire micro-program, as opposed to being limited to the first  $2^n$  instructions, where  $n$  is the width, in bits, of the address portion of the instruction register **40**. In order to utilize this approach, the control store **46** may include a secondary addressing scheme to associate “jump to” addresses to widely dispersed primary physical address locations in the store. Primary addressing in the control store **46** would still need to be maintained for use by the program counter **18** and also for updating the program counter **18** when a location is “jumped to.” For example, a word in the control store **46** could have a primary physical address of 10 and a secondary “jump to” address of 1. If the state of the processor **36** dictates a jump to “jump to” address 1, then the program counter **18** would need to be updated to 10, or the actual primary physical address of “jump to” address 1. The previously mentioned solution, however, in which the address and control/state portions of the instruction register **40** are utilized, is the preferred solution.

**[00126]** A distinct characteristic of the CNF Boolean processor **36** is the type of expressions it is designed to evaluate; namely expressions in CNF. Optionally, using a similar register design, a DNF-based architecture can also be implemented, as described herein below. However, the architecture of the CNF Boolean processor **36** focuses on CNF, providing the fastest and most scalable design.

**[00127]** The architecture of the DNF Boolean processor **136** is illustrated in Fig. 5. For the purposes of describing the architecture of the DNF Boolean processor **136**, 8-bit device addressing and 8-bit control words are used. This results in the architecture of the DNF Boolean processor **136** supporting 256 devices, each device having 256 possible states. Optionally, the architecture of the DNF Boolean processor **136** can be scaled to accommodate  $2^n$  devices, each device having  $2^m$  possible states, where  $n$  and  $m$  are the number of device address bits and the number of possible states for each device, respectively. The defining feature of the architecture of the DNF Boolean processor **136** is its set of registers, or lack thereof. In contrast to conventional microprocessors and microcontrollers, which can have a plurality of registers (typically from 8 to 64 bits wide), the DNF Boolean processor **136** has only six registers. Of the six registers, the instruction register **140**, the end of operation address register **142**, and the end of AND address register **144** are the only registers which are generally required to be multi-bit registers. The remaining three registers **154**, **156**, **158** hold AND truth states, OR truth states, and an indicator for disjuncts containing AND clauses. Each of these registers **154**, **156**, **158** may be only a single bit in size, although additional bits may be included if desired.

**[00128]** The DNF Boolean processor **136** includes the instruction register **140**, which is an  $n+m+x$ -bit wide register containing an  $n$ -bit address, an  $m$ -bit control/state word, and an  $x$ -bit operational code. Using 8-bit device addressing, 8-bit control words, and 3-bit operational codes, the instruction register **140** is 19 bits wide. The DNF Boolean processor **136** also includes a control store (ROM) **146**, which is used to hold a compiled micro-program, including  $(n+m+x)$ -bit instructions. The DNF Boolean processor **136** further includes the program counter **118**, which is used for fetching the next instruction from the control store **146**. The DNF Boolean processor **136** further includes a memory (MUX) **148**, which is used to configure the program counter **118** for normal operation, conditional jump operation,

unconditional jump operation, and Boolean short-circuit operation. Six AND gates **150** are used to pass operation results and a plurality of signals that are operational code dependent.

**[00129]** The OR register **154** is used to roll up the results of the disjuncts. If the OR register **154** is one bit in size, then the default value of the OR register **154** is zero and it initializes to a value of zero after a start of operational code. The 1-bit OR register **154** remains at a value of zero if all of the disjuncts in the Boolean expression being evaluated are false. If this bit is set to one at any time during the evaluation, the entire DNF operation is true. In such a case, the remainder of the operation may be short-circuited and the control operation that occurs as the result of a true evaluation can be executed. It should be apparent, however, that the OR register **154** may be modified such that one or more alternative values may be used to initialize the register **54** and represent a “false” value. The same applies to a “true” value as well, where any of another set of values (provided that the selected value is different from the one(s) used to represent a “false” value) may be used to represent a “true” value.

**[00130]** The AND register **156** is used to roll up the results of each of the individual disjuncts. If the AND register **156** is one bit in size, then it initializes to a value of one and remains in that state until a state in a disjunct evaluates to false. The AND disjunct register **158** is used to indicate that the evaluation of a disjunct containing AND clauses has begun. It initializes to a value of zero and remains in that state until an AND operation sets its value to one. It should be apparent, however, that the AND register **156** may be modified such that one or more alternative values may be used to initialize the register **156** and represent a “true” value. The same applies to a “false” value as well, where any of another set of values (provided that the selected value is different from the one(s) used to represent a “true” value) may be used to represent a “false” value. Finally, if the AND disjunct register **158** is one bit in size, then it initializes to a value of zero and remains in that state until an AND operation sets its value to one. In the event that the 1-bit AND disjunct register **158** is set to one and the 1-bit AND register **156** is set to zero, the entire disjunct evaluates to false and short-circuits to the start of the next disjunct.

**[00131]** The DNF Boolean processor 136 further includes an operation decoder 160, which deciphers each operational code and controls the units that are dependent upon each operational code. In an embodiment preferred for its simplicity, the operational codes are 3 bits in length, and the functions of the operation decoder 60 by operational code include: Boolean OR (Op Code 0), Boolean AND (Op Code 1), End of Operation (Op Code 2), No Operation (Op Code 3), Unconditional Jump (Op Code 4), Conditional Jump (Op Code 5), Start of Operation (Op Code 6), and Start of AND Disjunct (Op Code 7). However, it will be apparent that the inclusion of one or more additional bits in the instruction register 140 would permit additional operational codes to be offered, and that the removal of a bit would reduce the number of operational codes offered, if either such design were to be desired.

**[00132]** A control encoder 162 accepts  $n+m$  bits in parallel (representing a device address and control word) and outputs them across a device bus (control lines) either serially or in parallel, depending upon the architecture of the given device bus. The end of operation address register 142 stores the address used for Boolean short-circuiting. Short-circuiting occurs as soon as a disjunct evaluates to true. In such a case, the address is the address of the final control portion of the expression which results in the event that the entire DNF expression is true. The end of AND address register 144 stores the address of the instruction immediately following a disjunct containing AND clauses. It is used for the short-circuiting of disjuncts that contain AND clauses. The DNF Boolean processor 136 further includes a device state storage (RAM) 164, which is responsible for storing the states of the devices that the DNF Boolean processor 136 monitors and/or controls. It has  $2^n$  addresses, each of which are  $m$ -bits wide, where  $n$  is the address width and  $m$  is the control/state word width, in bits.

**[00133]** The DNF Boolean processor 136 evaluates micro-programs and controls its environment based upon the results of the above-described evaluations. The micro-programs define the actions to be taken by devices in the event that the given Boolean tests evaluate to true. The DNF Boolean processor 136 works on the principle that the devices will be controlled based upon their states and the states of other devices, or after some period of time has elapsed. In order to evaluate a micro-program, conditional tests must be compiled into Boolean Disjunctive Normal Form (DNF).

**[00134]** The DNF Boolean processor **136** performs eight functions, as specified by operational code. Op Code 0 – (Boolean OR) enables the AND gate **150** that loads the OR register **154** in the event that the conditional state of the device at the address in the instruction register **140** equals the state being tested in the instruction register **140**. The Boolean OR instruction serves two purposes. First, the instruction is used to rollup results between AND disjuncts. This is accomplished by comparing a “zero” value to the value in location 0, which always results in a “true” evaluation. Secondly, the instruction may be used to evaluate stand-alone disjuncts, in which case a value is being compared to a device state. Op Code 1 – (Boolean AND) sets the value of the AND disjunct register **158** to one, which enables short-circuiting within a disjunct containing AND clauses. Op Code 2 – (End of Operation) enables the AND gate **150** that passes the value of the AND register **156** to the OR register **154**. If the OR register **154** ever evaluates to a value of one, the program is short-circuited to the end of operation instruction (the control operation that executes in the event of a true evaluation) and the control encoder **162** is enabled and the address and control word specified in the end of operation code is sent to the proper device. Op Code 3 – (No Operation) does nothing. Op Code 4 – (Unconditional Jump) allows the MUX **148** to receive an address from the address portion of the instruction register **140** and causes an immediate jump to the instruction at that address. Op Code 5 – (Conditional Jump) provides that if the OR register **154** has a value of one, the test condition is met and the MUX **148** is enabled to receive the “jump to” address from the address portion of the instruction register **140**. Op Code 6 – (Start of Operation) provides the address of the end of operation line for the current operation. This address is used to short-circuit the expression and keep the DNF Boolean processor **136** from having to evaluate the entire DNF expression in the event that one of the disjuncts evaluates to one. In addition to loading the end of AND address into the end of AND address register **144**, this operation also sets the OR register **154** to zero, the AND register **156** to one and the AND disjunct register **158** to zero. Op Code 7 – (Start of AND Disjunct) provides the address of the line immediately following the disjunct and loads it into the end of AND address register **144**. This address is used to provide short-circuiting out of a given disjunct in the event that one of the disjunct’s terms evaluates to zero.

**[00135]** The evaluation of a DNF expression begins with Start of Operation (Op Code 6) and proceeds to the evaluation of a disjunct. A disjunct may be either a stand-alone term (evaluated as an OR operation) or a disjunct containing AND clauses. In the latter case, each term of the disjunct is evaluated as part of an AND operation (Op Code 1). Each of these operations represents a test to determine if the state of a given device is equal to the state value specified in the corresponding OR or AND instruction. If the term evaluates to false, the AND-bit is set to a value of zero. Otherwise, the AND-bit is set to a value of one. In the case of a stand-alone term, this value automatically rolls up to the OR register **154**. In disjuncts containing AND clauses, the result of each AND operation is AND'd with the current value of the AND register **156**. This ensures that a false term anywhere in the disjunct produces a final value of false for the entire disjunct evaluation. In the event that the AND register **156** has a value of zero and the AND disjunct register **158** is set to one, the disjunct will evaluate to false and may be short-circuited to the next disjunct. Next, the DNF Boolean processor **136** prepares for subsequent disjuncts (if any additional disjuncts exist). At this point, an OR operation (Op Code 0) joins the disjuncts and the value of the AND register **156** is rolled up to the OR register **154** by having the value of the AND register **156** passed through to the OR register **154**. In the event that the AND-bit has a value of one when the OR operation is processed, the OR-bit will change to a value of one. Otherwise, the OR-bit's value will remain at zero. If the OR-bit has a value of zero, the next disjunct is evaluated. If the OR-bit has a value of one, the final value of the DNF expression is true, regardless of the evaluation of any additional disjuncts. At this point, the remainder of the expression may be short-circuited and the end of operation instruction may be executed.

**[00136]** Preferably, the DNF Boolean processor **136** requires that functions be compiled in DNF. A micro-code compiler builds the micro-instructions such that they follow a DNF logic. The logic statements for DNF Boolean processor programs are nothing more than IF-THEN-ELSE statements. For example: IF (*Device A* has *State Ax*), THEN (Set *Device B* to *State By*), ELSE (Set *Device C* to *State Cz*). The logic of the IF expression must be compiled into DNF. The expression must also be expanded into a set of expressions OR'd together, and OR'd with a pre-set value of “false”. For the DNF operation, the pre-set value of “false” is the

initial value of the OR register **154** at the start of each logical IF operation. The above IF-THEN-ELSE statement would result in the following micro-code logic: [ (*Device A* has *State Ax*)  $\vee$  “false” ]; if the OR statement is “true”, then (SET *Device B* to *State By*); and if the OR statement is “false”, then (SET *Device C* to *State Cz*).

**[00137]** The following are examples of how some common operations would be compiled to work with the architecture of the DNF Boolean processor **136**. It should be noted that the Start of Operation Instruction (Op Code 6), as well as the Start of Disjunct Instruction (Op Code 7), have been omitted since ROM addresses are not listed in the examples. The notation in the following examples is of the form: DevX=Y, where X represents the device address and Y represents the current state of the device.

**[00138]      Example 3:**

*If (Dev1=8 and Dev2=0) or (Dev3=10 and Dev2=0) then Dev9=20*

Micro-code		
Sequence #	Value of Instruction Register	Operation
1	00000001 00001000 001	AND
2	00000010 00000000 001	AND
3	00000000 00000000 000	OR
4	00000011 00001010 001	AND
5	00000010 00000000 001	AND
6	00001001 00010100 010	End of Operation

**[00139]      Example 4:**

*If Dev7=22 goto 200*

*Else goto 100*

Micro-code		
Sequence #	Value of Instruction Register	Operation
1	00000111 00010110 001	AND
2	00000000 00000000 000	OR
3	00000000 11001000 101	Conditional Jump
4	00000000 01100100 100	Unconditional Jump

**[00140]** Once again, as illustrated in Fig. 5, the end of operation address register 142 and the end of AND address register 144 may be loaded with values from the  $n$ -bit “address” portion of the instruction register 140. However, in order to expand the micro-program address values that can be stored in the end of operation address register 142 and the end of AND address register 144, the architecture may be modified to use the bits from both the address and control/state portions of the instruction register 140 when loading the end of operation address register 142 and the end of AND address register 144 with the values of micro-program addresses. This approach would require the “control/state” portion of the instruction register 140 to be connected directly to the address registers 142, 144 in addition to the MUX 148. Further, as with the CNF Boolean processor 36, another solution is to modify the control store portion of the architecture to include discrete “jump to” addresses that would only be utilized on instructions that are capable of being jumped to, as described previously.

**[00141]** A distinct characteristic of the DNF Boolean processor 136 is the type of expressions it is designed to evaluate; namely expressions in DNF. It should be noted that the DNF Boolean processor 136 performs both inter and intra-term short-circuit evaluations, thereby providing maximum efficiency in processing expressions.

**[00142]** Upon initial inspection of the two forms, CNF and DNF, an individual might be inclined to believe that the short-circuit evaluation of DNF expressions has benefits over short-circuited CNF expressions because the terms are OR’d together and a positive result for any of the terms results in a completed evaluation. The same argument, in the false case, can be made for CNF evaluations. If any of the terms results in a false value, the entire evaluation is

complete with a value of false. Additionally, CNF eliminates repeating terms, as shown in the following examples.

**[00143]      Example 5:**

Conjunctive Normal Form

*If Dev2=0 and (Dev1=8 or Dev3=10 or Dev4=1) then Dev9=20*

Micro-code		
Sequence #	Value of Instruction Register	Operation
1	00000010 00000000 000	AND
2	00000001 00001000 001	OR
3	00000011 00001010 001	OR
4	00000100 00000001 001	OR
5	00001001 00010100 010	End of Operation

**[00144]      Example 6:**

Disjunctive Normal Form

*If (Dev2=0 and Dev1=8) or (Dev2=0 and Dev3=10) or (Dev2=0 and Dev4=1) then Dev9=20*

Micro-code		
Sequence #	Value of Instruction Register	Operation
1	00000010 00000000 000	AND
2	00000001 00001000 000	AND
3	00000000 00000000 001	OR
4	00000010 00000000 000	AND
5	00000011 00001010 000	AND

---

6	00000000 00000000 001	OR
7	00000010 00000000 000	AND
8	00000100 00000001 000	AND
9	00001001 00010100 010	End of Operation

---

**[00145]** Notice, in the examples, that the testing of Dev2 is a single conjunct in the CNF expression and repeated in every disjunct in the DNF expression. This type of term is important as the outcomes of both the CNF and DNF expressions are almost fully dependent upon their values. These terms are referred to herein as “control states” or “control devices”. Without a positive evaluation of a control state, any Boolean expression, whether in CNF or DNF, will evaluate to false. In the case of CNF, the false evaluation of a control state enables short-circuiting, and is what provides CNF with its advantage over DNF.

**[00146]** In the previous examples, CNF provides a savings of four instructions over DNF. DNF, however, has an advantage over CNF for a very small number of non-control, or “other” states (one or two). As the number of terms (both control and “other”) grows, however, the short-circuiting of CNF expressions becomes a much more efficient means of evaluation.

**[00147]** Two types of short-circuiting exist in CNF and DNF operations, inter-term short-circuiting and intra-term short-circuiting. Inter-term short-circuiting causes the evaluation of an entire expression to evaluate to true, in the case of DNF, or false, in the case of CNF, if any term evaluates to true or false, respectively. Intra-term short-circuiting causes the evaluation of a conjunct or disjunct to terminate without full evaluation. In this instance, a CNF term, or conjunct, will evaluate to true if any of its sub-terms are true, while a DNF term, or disjunct, will evaluate to false if any of its sub-terms are false. Consider the following statements:

$$\text{CNF: } \text{If } (A \text{ or } B) \text{ and } (C \text{ or } D) \text{ then } E \quad (7)$$

$$\text{DNF: } \text{If } (A \text{ and } B) \text{ or } (C \text{ and } D) \text{ then } E \quad (8)$$

**[00148]**

In the CNF statement, if A evaluates to true, the entire conjunct A or B evaluates to true. As a result, the evaluation of B is unnecessary and can be avoided using intra-term short-circuit evaluation. From an inter-term perspective, if the conjunct A or B evaluates to false, the entire CNF expression evaluates to false, making the evaluation of the conjunct C or D superfluous. In the case of DNF, both inter and intra-term short-circuit evaluation work similarly to that of CNF, except that the term values for DNF are the converse of those for CNF. It should be noted that the Boolean processors 36, 136 perform both inter and intra-term short-circuit evaluations, thereby providing maximum efficiency in processing expressions.

**[00149]**

In examining the inter-term short-circuit evaluation of both CNF and DNF expressions, the following equations can be used to characterize the behavior of each:

*Avg. CNF Instructions =*

$$((ICS * CS) + (IOS * OS)) * PCSD + (ICS * CS) * (1 - PCSD) * FCSD \quad (9)$$

*Avg. DNF Instructions =*

$$((ICS * CS) + IOS) * OS * (PCSD * POSD + (1 - PCSD)) \quad (10)$$

where: ICS = number of processor instructions required to process a control state; CS = number of control states; OS = other, or non-control, states; IOS = number of processor instructions required to process an “other” state; PCSD = positive control state distribution, the probability that all control states evaluate to true (e.g., a PCSD of .5 means that all of the control states evaluate to true in fifty percent of the expression evaluations); FCSD = false control state distribution, in the event that the control states evaluate to false, this number represents which of the control states caused the failure (e.g., a failure among 10 control states with an FCSD of .7 means the 7<sup>th</sup> control state evaluated to false); POSD = positive “other” state distribution, the position within the expression that an “other” state evaluates to true (e.g., a POSD of .5 means the 5<sup>th</sup> term of 10 evaluates to true).

**[00150]**

The following charts represent the results of varying the number of control states and “other” states in the above-referenced equations. It should be noted that all control

states are evaluated as soon as possible (i.e. moved as far left in the expression as possible). In this manner, the control states are the first conjuncts in CNF equations and the first terms in each disjunct of DNF equations. Additionally, in the case of DNF equations, each “other” state is combined with the control states to form a disjunct. This results in an equal number of “other” states and disjuncts. Data is generated using a CNFDNF emulation program and complementary CNF and DNF expression classes. A fixed number of control states is entered for each run of the program. The program then varies the number of “other” states from zero to one-thousand, for example. At each step, a random POSD (between 0 and 1) is used and averaged over one-million iterations.

**[00151]** Fig. 6 illustrates the limited advantage of DNF. Only when the probability for the control state to evaluate to true is one-hundred percent, or the number of “other” states is less than two at a seventy-five percent control state probability, does DNF have an advantage over CNF. When the control state probability is one-hundred percent, meaning the control states are always true, and the number of “other” states becomes arbitrarily large, DNF retains its slight advantage over CNF, as illustrated in Fig. 7. This advantage becomes trivial as the difference in number of processor instructions is minimal at this small number of states. In addition, the likelihood of a one-hundred percent probability for a control state is very remote. For example, in the case of a home alarm system. An alarm will sound if the system is armed and a door or window is opened. In this case, the system being armed is the control state. In a real-world application, however, an alarm system is not continuously armed. Once the number of “other” states rises above two, CNF has a distinct advantage over DNF. This advantage becomes greater as the number of control states and “other” states rises, and the control state probability drops. This is apparent in Figs. 6 through 11.

**[00152]** As the number of “other” states becomes arbitrarily large, the ratio of DNF evaluations to CNF evaluations becomes relatively constant. Taking a closer look at the formulas for DNF and CNF instructions as OS becomes relatively large and PCSD becomes relatively small, DNF becomes a function of  $(OS * CS)$ , while CNF becomes a function of  $(OS * PCSD)$ . Thus, the DNF to CNF instruction ratio can be expressed as an approximate function of the number of control states and their positive distribution, or hit rate, such that DNF/CNF Ratio

$\approx CS / PCSD$ . Because a relatively large number of control states usually corresponds to a relatively low probability, the choice of CNF over DNF becomes advantageous as the size of the system grows.

**[00153]** The combination of inter and intra-term short-circuiting provides a significant performance gain over the use of either one alone. Assuming that only one of x “other” states will evaluate to true during any single evaluation of an expression, the addition of intra-term short-circuiting reduces the number of state evaluations by (.5 \* # of “Other” States) and (.5 \* # of Control States \* # of “Other” States) on average for CNF and DNF expressions, respectively. Using both inter and intra-term short-circuiting, the above-referenced equations given to describe the average number of instructions for both CNF and DNF become:

$$\begin{aligned} \text{Avg. CNF Instructions} = & ((ICS * CS) + (IOS * OS * POSD)) \\ & * PCSD + (ICS * CS) * (1 - PCSD) * FCSD \end{aligned} \quad (11)$$

$$\begin{aligned} \text{Avg. DNF Instructions} = & ((ICS * CS + IOS) * OS * PCSD * POSD) \\ & + ((ICS * CS * (1 - PCSD) * FCSD + IOS) * OS \end{aligned} \quad (12)$$

Figs. 12 through 15 illustrate the DNF/CNF ratio for 1 and 10 control states, each combined with small and large numbers of “other” states.

**[00154]** Using inter and intra-term short-circuiting together ultimately results in the identical DNF/CNF ratio (for large “other” states) as when using only inter-term short-circuiting. However, the number of average evaluations for each of the two Boolean forms is reduced by fifty percent. Prior to reaching the ratio limit, the effect of using both types of short-circuiting on DNF is especially prevalent, as illustrated by the reduction of the slope of the curve of Fig. 15 as compared with its inter-term short-circuiting counterpart of Fig. 11.

**[00155]** Thus, short-circuiting provides a performance gain by reducing the number of instructions evaluated by the Boolean processor **36** (Figs. 3 and 4). The actual gain is a function of the number of control states, the number of “other” states, the number of

instructions for each, and the positive distribution of each. Using the above-referenced equation for the average CNF evaluations for inter and intra-term short circuiting:

$$\begin{aligned}
 \text{Avg. CNF Instructions} &= ((ICS * CS) + (IOS * OS * POSD)) \\
 &\quad * PCSD + (ICS * CS) * (1 - PCSD) * FCSD
 \end{aligned} \tag{13}$$

and the following formula for the number of evaluations for non-short-circuited CNF:

$$\text{CNF Instructions} = (ICS * CS) + (IOS * OS) \tag{14}$$

the improvement that short-circuiting provides can be evaluated, as illustrated in Table 1.

Control States	"Other" States	Positive Control State Distribution	Positive "Other" State Distribution	Average Short-Circuit Evaluations (FCSD = 0.5)	Non-Short-Circuit Evaluations	% Savings Using Short-Circuiting
1	5	1	0.5	3.5	6	41.67%
1	10	1	0.5	6	11	45.45%
1	10	0.5	0.5	3.25	11	70.45%
1	10	0.25	0.5	1.875	11	82.95%
1	10	0.1	0.5	1.05	11	90.45%
1	100	1	0.5	51	101	49.50%
1	100	0.5	0.5	25.75	101	74.50%
1	100	0.25	0.5	13.125	101	87.00%
1	100	0.1	0.5	5.55	101	94.50%
1	1000	1	0.5	501	1001	49.95%
1	1000	0.5	0.5	250.75	1001	74.95%
1	1000	0.25	0.5	125.625	1001	87.45%
1	1000	0.1	0.5	50.55	1001	94.95%
5	5	1	0.5	7.5	10	25.00%
5	10	1	0.5	10	15	33.33%
5	10	0.5	0.5	6.25	15	58.33%
5	10	0.25	0.5	4.375	15	70.83%
5	10	0.1	0.5	3.25	15	78.33%
5	100	1	0.5	55	105	47.62%
5	100	0.5	0.5	28.75	105	72.62%
5	100	0.25	0.5	15.625	105	85.12%
5	100	0.1	0.5	7.75	105	92.62%
5	1000	1	0.5	505	1005	49.75%
5	1000	0.5	0.5	253.75	1005	74.75%
5	1000	0.25	0.5	128.125	1005	87.25%
5	1000	0.1	0.5	52.75	1005	94.75%
10	10	1	0.5	15	20	25.00%

10	20	1	0.5	20	30	33.33%
10	10	0.5	0.5	10	20	50.00%
10	10	0.25	0.5	7.5	20	62.50%
10	10	0.1	0.5	6	20	70.00%
10	100	1	0.5	60	110	45.45%
10	100	0.5	0.5	32.5	110	70.45%
10	100	0.25	0.5	18.75	110	82.95%
10	100	0.1	0.5	10.5	110	90.45%
10	1000	1	0.5	510	1010	49.50%
10	1000	0.5	0.5	257.5	1010	74.50%
10	1000	0.25	0.5	131.25	1010	87.00%
10	1000	0.1	0.5	55.5	1010	94.50%

Table 1 – Percentage of Instructions Saved Using CNF Short-Circuited Evaluation Instead of Non-Short-Circuited Evaluation

**[00156]** Because the number of instructions required to evaluate a control state is typically the same as the number required to evaluate “other” states, one instruction is assumed for each. The savings illustrated in the Table 1 range from twenty-five to almost ninety-five percent. While the high-end of this range represents a typical system, in terms of the number of control states versus “other” states, the low-end of the range occurs when the number of control states is equal to or near the number of “other” states. In a typical configuration, the number of “other” states outweighs the number of control states, resulting in a relatively higher instruction evaluation savings. In light of all of the above, the use of CNF outweighs any benefit provided by DNF, thereby warranting an architectural design that uses Boolean expressions compiled into CNF.

**[00157]** Although, as described above, the overall processing efficiency for CNF expressions is generally greater than that of DNF expressions, it still may be advantageous to be able to efficiently process either type of expression in certain computing environments. In such situations, a combined CNF/DNF processor (not shown) may be implemented by combining the common portions of the respective processors disclosed in Figs. 4 and 5 with appropriate combinatorial logic for controlling their operation according to the mode in which the combined processor is to be operating. By providing a switch or control means for controlling the operation mode, such a combined processor would encompass the benefits of both the CNF processor 36 and the DNF processor 136 by operating in the mode appropriate for the type of

expression being evaluated. The switch or control means may be controlled by the use of an extra instruction or operation code. The design and control of such a combined processor would otherwise be apparent to one of ordinary skill in the art.

**[00158]** As described above, the Boolean processor 36 (Figs. 3 and 4) is a general-purpose architecture that is intended to monitor and control its surrounding environment. The small instruction set and design of the Boolean processor 36 are intended to give it a significant speed advantage over its competition, namely other general-purpose architectures, such as microprocessors and microcontrollers. To demonstrate this advantage, the instruction set of the Boolean processor 36 is compared with the instruction sets of two other general-purpose architectures; the Intel 8051 microcontroller and the Intel 8088 family of microprocessors. In addition, the Motorola MMC2107 microcontroller is used for comparison purposes.

**[00159]** Like other microcontrollers, the main purpose of the 8051 is to control its surrounding environment. Because the 8051 is not optimized for Boolean operations, it requires the use of several instructions in order to emulate the functions of the Boolean processor 36 of the present invention. In addition, it also requires the use of two registers: one register to hold the intermediate results of OR calculations and another register for retrieving device states from memory. AND calculations resulting in a false value can be handled by issuing a jump past the operation that results from a true evaluation of the statement. The instructions required to perform the same operations as those of the Boolean processor 36 are illustrated in Table 2. It should be noted that the label SHORT is the label for the instruction immediately following the current CNF expression and is used for inter-term short-circuiting. The SHORTCON label is the label for the next OR term of a conjunct and is used for intra-term short-circuiting.

Boolean Processor Instruction	8051 Equivalent Instruction(s)	Explanation
AND	MOV Rr, add CJNE Rr, #n, SHORT	Move the state of the device at add into register Rr, If the value in Rr is not equal to the state specified in #n, then the expression is false and may be short circuited.

---

OR	MOV Rr, add CJNE Rn, #n, NEXT MOV Rx, #1 SJMP SHORTCON NEXT MOV Rx, #0	Move the state of the device at add into register Rr. If the value in Rr is not equal to the state specified in #n, we set Rx to 0 and move on to the next OR term or the next conjunct. If it is equal, we set the value of Rx to 1 and short circuit the remainder of the conjunct. Rx will be used at the end of operation to determine the final outcome of the entire CNF expression.
End of Operation	CJNE Rx, #1, SHORT MOV #n, add	If Rx does not equal one, then one of the conjuncts containing OR terms did not evaluate to true. Therefore, the entire expression is false. If Rx is equal to one, the expression is true and the resulting state change may be made to the device at add.
No Operation	NOP	PC = PC +1
Unconditional Jump	SJMP radd	Jump to the line specified by radd.
Conditional Jump	CJNE Rn, #n, radd	If the value in Rn is not equal to #n, then jump to the line specified by radd.
Start of Operation	MOV Rx, #1	Move 1 into register Rx which is used to hold the result value of conjuncts containing OR terms.
Start of OR Conjunct	MOV Rx, #1	See Start of Operation.
	CJNE Rx, #1, SHORT	This operation is required at the end of an OR Conjunct. If the value of Rx is not equal to 1, the value of the conjunct and, hence, the value of the entire expression is false and may be short-circuited.

---

Table 2 – Intel 8051 Equivalent Instructions

**[00160]      Example 7:**

The statement: *If dev1=1 and dev2=3 and (dev3=1 or dev4=2) then dev6=8*, is written for the Intel 8051 as follows:

---

BEGIN:	MOV R1, #1	Initialize R1 to 1 (OR holder)
	MOV R0, dev1	Move state of device 1 into R0
	CJNE R0, #1, SHORT	If dev1 not = to 1 then goto

---

---

	MOV R0, dev2	SHORT Move state of device 2 into R0
	CJNE R0, #3, SHORT	If dev2 not = to 3 then goto SHORT
	MOV R0, dev3	Move state of device 3 into R0
	CJNE R0, #1, NEXT	If dev3 not = to 1 then goto NEXT
	MOV R1, #1	If true set R1 to 1
	SJMP SHORTCON	Jump to end of OR conjunct
NEXT:	MOV R1, #0	If false set R1 to 0
	MOV R0, dev4	Move state of device 4 into R0
	CJNE R0, #2, NEXT2	If dev4 not = to 2 then goto NEXT2
	MOV R1, #1	If true set R1 to 1
	SJMP SHORTCON	Jump to end of OR conjunct
NEXT2:	MOV R1, #0	If false set R1 to 0
SHORTCON:	CJNE R1, #1, SHORT	If OR Holder not = 1 then goto SHORT
	MOV #8, dev6	Expression is true, set dev6 = 8
SHORT:	NOP	No Operation or start of next if then block
	SJMP BEGIN	Start again and keep testing

---

**[00161]** The same statement is implemented for the Boolean processor 36 using the following code:

---

Instruction Register Address				
Control Store Address	Address	Control/State	Opcode	Functionality
00000000	00000011	00000000	110	Start of Boolean expr.
00000001	00000001	00000001	000	Dev1 = 1? (AND)

00000010	00000010	00000011	000	Dev2 = 3? (AND)
00000011	00000110	00000000	111	Start of OR conjunct
00000100	00000011	00000001	001	Dev3 = 1? (OR)
00000101	00000100	00000010	001	Dev4 = 2? (OR)
00000110	00000110	00001000	010	End of Operation
00000111	00000000	00000000	100	Jump to beginning

---

**[00162]** What required eighteen instructions using the 8051, required only eight instructions using the Boolean processor **36**. Using the differences in the number of instructions required for each operation, the extra number of instructions required to emulate the functionality of the Boolean processor **36** for an 8051 can be measured as such:

$$\text{Extra Instructions} = D\text{-And} * CS + D\text{-Or} * OS + OC + D\text{-EoO} \quad (15)$$

where: D-And = difference in number of instructions for an And operation = 1; CS = number of control states; D-Or = difference in number of instructions for an Or Operation = 4; OS = number of “other” states; OC = number of OR conjuncts; and D-EoO = difference in number of instructions for an End of Operation = 1. Simplified, the resulting equation is:

$$\text{Extra Instructions} = CS + 4OS + OC + 1 \quad (16)$$

**[00163]** The two jump codes, the two start codes, and the no-op code are not included in the calculation because they all require one instruction on each architecture and would, therefore, cancel out with a difference of zero. The number of OR conjuncts is taken into account since the 8051 requires an extra instruction to handle each one. Assuming, that as the size of system grows, the number of “other” states grows exponentially relative to the number of control states and the number of OR conjuncts; the number of extra instructions becomes a linear function such that: Extra Instructions = 4OS. This difference becomes significant as the number of “other” states becomes relatively large, as illustrated in Fig. 16.

**[00164]** The Intel 8086 family of microprocessors includes upward-compatibility which allows code written for previous-generation chips to be run on its ancestors. The 8086

family includes the 8086, 80186, 80286, 80386, 80486, and the Pentium models, each offering enhancements to that of its predecessor in terms of performance, memory management, and, in some cases, instruction sets. The basic jump, test, and move instructions required to emulate the functionality of the Boolean processor 36 are part of each of the processor's basic instruction set and can be used to represent the entire family. Being general-purpose platforms, the Intel microprocessors, like the 8051, are not optimized for Boolean operations. As a result, they also require the use of two registers for holding the results of OR operations and for storing states retrieved from memory. The instructions required to perform the same operations as those of the Boolean processor are illustrated in Table 3.

Boolean Processor Instruction	8086 Family Equivalent Instruction(s)	Explanation
AND	MOV Rr, add CMP Rr, #n JNE SHORT	Move the state of the device at add into register Rr. If the value in Rr is not equal to the state specified in #n, then the expression is false and may be short circuited.
OR	MOV Rr, add CMP Rn, #n JNE NEXT MOV Rx, #1 JMP SHORTCON NEXT MOV Rx, #0	Move the state of the device at add into register Rr. If the value in Rr is not equal to the state specified in #n, we set Rx to 0 and move on to the next OR term or the next conjunct. If it is equal, we set the value of Rx to 1 and short circuit the remainder of the conjunct. Rx will be used at the end of operation to determine the final outcome of the entire CNF expression.
End of Operation	CMP Rx, #1 JNE SHORT MOV #n, add	If Rx does not equal one, then one of the conjuncts containing OR terms did not evaluate to true. Therefore, the entire expression is false. If Rx is equal to one, the expression is true and the resulting state change may be made to the device at add.

No	NOP	PC = PC +1
Operation Uncondition al Jump	SJMP radd	Jump to the line specified by radd.
Conditional Jump	CMP Rn, #n JNE radd	If the value in Rn is not equal to #n, then jump to the line specified by radd.
Start of Operation	MOV Rx, #1	Move 1 into register Rx which is used to hold the result value of conjuncts containing OR terms.
Start of OR Conjunct	MOV Rx, #1	See Start of Operation.
-	CMP Rx, #1 JNE SHORT	This operation is required at the end of an OR Conjunct. If the value of Rx is not equal to 1, the value of the conjunct and, hence, the value of the entire expression is false and may be short-circuited.

Table 3 – Intel 8086 Family of Microprocessors Equivalent Instructions

**[00165]      Example 8:**

The statement given in the previous Intel 8051 example:

*If dev1=1 and dev2=3 and (dev3=1 or dev4=2) then dev6=8*

would be written for an 8086 processor as follows:

BEGIN:	MOV R1, #1	Initialize R1 to 1 (OR holder)
	MOV R0, dev1	Move state of device 1 into R0
	CMP R0, #1	Compare dev1 state to 1
	JNE R0, #1, SHORT	If dev1 not = to 1 then goto SHORT
	MOV R0, dev2	Move state of device 2 into

---

	CMP R0, #3	R0
	JNE SHORT	Compare dev2 state to 3 If dev2 not = to 3 then goto SHORT
	MOV R0, dev3	Move state of device 3 into R0
	CMP R0, #1	Compare dev3 state to 1
	JNE NEXT	If dev3 not = to 1 then goto NEXT
	MOV R1, #1	If true set R1 to 1
	JMP SHORTCON	Jump to end of OR conjunct
NEXT:	MOV R1, #0	If false set R1 to 0
	MOV R0, dev4	Move state of device 4 into R0
	CMP R0, #2	Compare dev4 state to 2
	JNE NEXT2	If dev4 not = to 2 then goto NEXT2
	MOV R1, #1	If true set R1 to 1
	JMP SHORTCON	Jump to end of OR conjunct
NEXT2:	MOV R1, #0	If false set R1 to 0
SHORTCON:	CMP R1, #1	Compare OR Holder to 1
	JNE SHORT	If OR Holder not = 1 then goto SHORT
	MOV #8, dev6	Expression is true, set dev6 = 8
SHORT:	NOP	No Operation or start of next if then block
	JMP BEGIN	Start again and keep testing

---

**[00166]** In the above-referenced example, the 8086 family requires twenty-four instructions to execute the same functionality that only requires eight instructions for the Boolean processor **36**. Using the differences in the number of instructions required for each operation, the extra number of instructions required to emulate the functionality of the Boolean processor **36** for the 8086 can be measured as such:

$$\text{Extra Instructions} = D\text{-And} * CS + D\text{-Or} * OS + CJ + D\text{-OC}$$

$$* OC + D-EoO \quad (17)$$

where: D-And = difference in number of instructions for an And operation = 1; CS = number of control states; D-Or = difference in number of instructions for an Or Operation = 4; OS = number of “other” states; CJ = number of conditional jumps (difference = 1); D-OC = difference in number of instructions for an Or Conjunct = 2; OC = number of OR conjuncts; and D-EoO = difference in number of instructions for an End of Operation. Simplified, the resulting equation is:

$$\text{Extra Instructions} = 2CS + 5OS + CJ + 2OC + 2 \quad (18)$$

The unconditional jump code, the two start codes, and the no-op code are not included in the calculation because they all require one instruction for each architecture and would, therefore, cancel out with a difference of zero. Assuming that as the size of system grows, the number of “other” states grows exponentially relative to the number of control states and the number of OR conjuncts, the number of extra instructions becomes a linear function such that: Extra Instructions = 5OS, as illustrated in Fig. 17.

**[00167]** The Motorola MMC2107 is a microcontroller that is designed to meet the needs of distribution channel customers dealing with applications, such as vending machines, building management and heating-ventilation-air conditioning (HVAC) systems, exercise equipment and lighting control. Similar to the comparisons of the Boolean processor 36 to the 8051 and 8086 family, the emulation of the Boolean processor 36 by the MMC2107 requires the use of two registers for holding the results of OR operations and for storing states retrieved from memory. The instructions required to perform the same operations as those of the Boolean processor 36 are illustrated in Table 4.

Boolean Processor Instruction	Motorola MMC2107 Equivalent Instruction(s)	Explanation
-------------------------------------	--	-------------

AND	LDB Rr, add	Load the state of the device at add into register Rr, If the value in Rr is not equal to the state specified in imm5, then the expression is false and may be short circuited. (This is used for up to 32 states).
	CMPNEI Rr, imm5 BT SHORT	
- or -		
	LDB Rr, add MOVI Rn, imm7 CMPNE Rr, Rn BT SHORT	Load the state of the device at add into register Rr, Put the value of the state being compared in Rn. If the value in Rr is not equal to the state in Rn, then the expression is false and may be short circuited. (This is used for up to 128 states).
- or -		
	LDB Rr, add MOVI Rn, imm7 LSLI Rn, imm5 ADDI Rn, oimm5 CMPNE Rr, Rn BT SHORT	Load the state of the device at add into register Rr, Put the value of the state being compared in Rn. In order to compare Rr to a number greater than 127, the value must be loaded into Rn via a series of Logical Left Shifts and Adds. If the value in Rr is not equal to the state in Rn, then the expression is false and may be short circuited. (This is used for states > 128).
OR	LDB Rr, add CMPNEI Rn, imm5 BT NEXT MOVI Rx, 1 JMPI SHORTCON	Move the state of the device at add into register Rr. If the value in Rr is not equal to the state specified in imm5, we set Rx to 0 and move on to the next OR term or the next conjunct. If it is equal, we set the value of Rx to 1 and short circuit the remainder of the conjunct. Rx will be used at the end of operation to determine the final outcome of the entire CNF expression. (This block is used for up to 32 states.)
	NEXT MOVI Rx, 0	
- or -		
	LDB Rr, add MOVI Rn, imm7 CMPNE Rr, Rn BT NEXT MOVI Rx, 1 JMPI SHORTCON	Move the state of the device at add into register Rr. Load the comparison value into Rn. If the value in Rr is not equal to Rn, we set Rx to 0 and move on to the next OR term or the next conjunct. If it is equal, we set the value of Rx to 1 and short circuit the remainder of the conjunct. Rx will be used at the end of operation to determine the final outcome of the entire CNF expression. (This block is used for up to 128 states.)
- or -		
	LDB Rr, add MOVI Rn, imm7 LSLI Rn, imm5 ADDI Rn, oimm5 CMPNE Rr, Rn BT NEXT MOVI Rx, 1 JMPI SHORTCON	Move the state of the device at add into register Rr. Load the comparison value into Rn. In order to compare Rr to a number greater than 127, the value must be loaded into Rn via a series of Logical Left Shifts and Adds. If the value in Rr is not equal to Rn, we set Rx to 0 and move on to the next OR term or the next conjunct. If it is equal, we set the value of Rx to 1 and short circuit the remainder of the conjunct. Rx will be used at the end of operation to determine the final outcome of the entire CNF expression. (This block is used for > 128 states.)
	NEXT MOVI Rx, 0	

End of Operation	CMPNEI Rx, 1 BT SHORT MOVI Rn, imm7 STB Rn, add	If Rx does not equal one, then one of the conjuncts containing OR terms did not evaluate to true. Therefore, the entire expression is false. If Rx is equal to one, the expression is true and the resulting state change may be made to the device at add. (This block is used for up to 128 state values.)
	- or -	
	CMPNEI Rx, 1 BT SHORT MOVI Rn, imm7 LSLI Rn, imm5 ADDI Rn, oimm5 STB Rn, add	If Rx does not equal one, then one of the conjuncts containing OR terms did not evaluate to true. Therefore, the entire expression is false. If Rx is equal to one, the expression is true and the resulting state change may be made to the device at add. In order to make a state change to a value higher than 127, the value must be loaded into Rn via a series of Logical Left Shifts and Adds. (This block is used for > 128 state values.)
No Operation	ADDI PC, 1	Increment the program counter by 1. PC = PC +1
Unconditional Jump	JMPI radd	Jump to the line specified by radd.
Conditional Jump	CMPNEI Rr, imm5 BT radd	If the value in Rr is not equal to imm5, then jump to the line specified by radd. (This block is used for comparison with values up to 32).
	- or -	
	MOVI Rr, imm7 CMPNE Rr, Rn BT radd	Place the comparison state value in Rn. If the value in Rr is not equal to the value in Rn, then jump to the line specified by radd. (This block is used for comparison with values up to 127).
	- or -	
	MOVI Rn, imm7 LSLI Rn, imm5 ADDI Rn, oimm5 CMPNE Rr, Rn BT radd	Place the comparison state value in Rn. In order to load Rn with a value higher than 127, the value must be loaded into Rn via a series of Logical Left Shifts and Adds. If the value in Rr is not equal to the value in Rn, then jump to the line specified by radd. (This block is used for comparison with values > 127).
Start of Operation	MOVI Rx, 1	Move 1 into register Rx which is used to hold the result value of conjuncts containing OR terms.
Start of OR Conjunct	MOVI Rx, 1	See Start of Operation.
-	CMPNEI Rx, 1 BT SHORT	This operation is required at the end of an OR Conjunct. If the value of Rx is not equal to 1, the value of the conjunct and, hence, the value of the entire expression is false and may be short-circuited.

Table 4 – Motorola MMC2107 Equivalent Instructions

**[00168]      Example 9:**

The statement given in the previous 8051 and 8086 family examples: *If dev1=1 and dev2=3 and (dev3=1 or dev4=2) then dev6=8*, would be written for the MMC2107 as follows:

---

BEGIN:	MOVI R1, 1	Initialize R1 to 1 (OR holder)
	LDB R0, dev1	Move state of device 1 into R0
	CMPNEI R0, 1	Compare dev1 state to 1
	BT SHORT	If dev1 not = to 1 then goto SHORT
	LDB R0, dev2	Move state of device 2 into R0
	CMPNEI R0, 3	Compare dev2 state to 3
	BT SHORT	If dev2 not = to 3 then goto SHORT
	LDB R0, dev3	Move state of device 3 into R0
	CMPNEI R0, 1	Compare dev3 state to 1
	BT NEXT	If dev3 not = to 1 then goto NEXT
	MOVI R1, 1	If true set R1 to 1
	JMPI SHORTCON	Jump to end of OR conjunct
NEXT:	MOVI R1, 0	If false set R1 to 0
	LDB R0, dev4	Move state of device 4 into R0
	CMPNEI R0, 2	Compare dev4 state to 2
	BT NEXT2	If dev4 not = to 2 then goto NEXT2
	MOVI R1, 1	If true set R1 to 1
	JMPI SHORTCON	Jump to end of OR conjunct
NEXT2:	MOVI R1, #0	If false set R1 to 0
SHORTCON:	CMPNEI R1, 1	Compare OR Holder to 1
	BT SHORT	If OR Holder not = 1 then goto SHORT
	MOVI R0, 8	Expression is true, set dev6 = 8
	STB	R0, dev6
SHORT:	NOP	No Operation or start of next if then block
	JMP BEGIN	Start again and keep testing

---

**[00169]** In the above-referenced example, the MMC2107 requires twenty-five instructions to execute the same functionality that only requires eight instructions for the Boolean processor **36**. It should also be noted that the MMC2107's M•CORE™ instruction set requires the use of additional instructions for loading and comparing values greater than thirty-two (see the "Explanation" column of Table 4). Using the differences in the number of instructions required for each operation, the extra number of instructions required to emulate the functionality of the Boolean processor **36** for a Motorola MMC2107 can be measured as such:

$$\begin{aligned} \text{Extra Instructions} &= D\text{-And} * CS + D\text{-Or} * OS + CJ + D\text{-OC} \\ &\quad * OC + D\text{-EoO} \end{aligned} \quad (19)$$

where: D-And = difference in number of instructions for an And operation; CS = number of control states; D-Or = difference in number of instructions for an Or Operation; OS = number of "other" states; CJ = number of conditional jumps; D-OC = difference in number of instructions for an Or Conjunct; OC = number of OR conjuncts; and D-EoO = difference in number of instructions for an End of Operation. Simplified, the resulting equations are:

$$\text{Extra Instructions} = 2CS + 5OS + CJ + 2OC + 3 \quad (\text{for } < 32 \text{ states}) \quad (20)$$

$$\text{Extra Instructions} = 3CS + 6OS + 2CJ + 2OC + 3 \quad (\text{for } \leq 128 \text{ states}) \quad (21)$$

and

$$\text{Extra Instructions} = 5CS + 8OS + 4CJ + 2OC + 5 \quad (\text{for } > 128 \text{ states}) \quad (22)$$

The unconditional jump code, the two start codes, and the no-op code are not included in the calculation because they all require one instruction on each architecture and would, therefore, cancel out with a difference of zero. Assuming that as the size of system grows, the number of "other" states grows exponentially relative to the number of control states and the number of OR conjuncts, the number of extra instructions becomes a series of linear functions such that:

$$\text{Extra Instructions} = 5\text{OS} \quad (\text{for } < 32 \text{ states}) \quad (23)$$

$$\text{Extra Instructions} = 6\text{OS} \quad (\text{for } \leq 128 \text{ states}) \quad (24)$$

$$\text{Extra Instructions} = 8\text{OS} \quad (\text{for } > 128 \text{ states}) \quad (25)$$

These functions are illustrated in Fig. 18. It will be apparent, however, that these values could change depending upon the size of the values being loaded and compared.

**[00170]** Still further efficiencies of Boolean processor technology, relative to conventional microcontrollers and microprocessors such as those described hereinabove, may be provided through the use of intelligent compiling or configuring when ordering terms, conjuncts, disjuncts and/or other operations. Fig. 19 is a flowchart diagram of a re-compiling process **1000** for use with the preferred embodiments of the present invention. This process may be used in conjunction with either a CNF Boolean processor **36** or a DNF Boolean processor **136**.

**[00171]** In a CNF Boolean processor **36**, the efficiency of the short circuiting of CNF expressions can be maximized by:

C1. Evaluating terms within conjuncts that are most likely to be true as early as possible in the overall evaluation of each conjunct.

C2. Evaluating conjuncts that are most likely to evaluate to false as early as possible in the overall evaluation of the CNF expression.

As shown in Fig. 19, the re-compiling process **1000** begins at step **1005** with an initial compiling of the code representing the Boolean expressions. The process **1000** then enters a loop which begins with the code actually being processed and the expressions themselves being evaluated at step **1010**. The next step **1015** in the loop is to determine (or update) the probabilities of terms within conjuncts evaluating to true and/or false and to store the updated probability info in some form in a memory. As the CNF expressions are evaluated over multiple iterations, the stored probabilities tend to become more accurate. When at step **1020** it is determined that a sufficient

amount of statistical data has been gathered and included in the calculation of probabilities, the process proceeds at step **1025** to re-compile the code representing the Boolean expressions in order to place it in an order likely to maximize the efficiency of the evaluations as described above in C1 and C2. This process **1000** may be repeated as often as desired or as often as is likely to improve the efficiency of the operation of the CNF Boolean processor **36**.

**[00172]** Similarly, in a DNF Boolean processor **136**, the efficiency of the short circuiting of DNF expressions can be maximized by:

D1. Evaluating terms within disjuncts that are most likely to be false as early as possible in the overall evaluation of each disjunct.

D2. Evaluating disjuncts that are most likely to evaluate to true as early as possible in the overall evaluation of the DNF expression.

The re-compiling process **1000** is the same as that for the CNF Boolean processor **36**, except that code represents DNF expressions that are evaluated and for which probabilities are determined before re-compiling the code in order to place it in an order likely to maximize the efficiency of the evaluations as described above in D1 and D2.

**[00173]** The comparisons provided previously between Boolean processors **36**, **136** and typical conventional microcontrollers or microprocessors were based on the use of a Boolean processor **36**, **136** instead of, or as a replacement for, the conventional microcontroller or microprocessor. However, in another aspect of the present invention, a Boolean processor **36**, **136** may be used in conjunction with another microcontroller or microprocessor. In this case, the Boolean processor **36**, **136** may function as a co-processor that is incorporated into larger chip designs such as other microcontrollers or microprocessors. Conceptually, this arrangement may serve a similar purpose to that of a math co-processor, except that the Boolean co-processor **336** would execute Boolean operations instead of mathematical operations, thus providing greater efficiency with regard to Boolean-intensive processing.

**[00174]** This may be accomplished in a variety of ways. For example, Fig. 20 is a high-level block diagram illustrating the functional relationship between a host computer device **308** and a separate Boolean co-processor **336**. The Boolean co-processor **336** comprises a CNF Boolean processor **36** and/or a DNF Boolean processor **136**, constructed as described previously but modified to accept portions of code from the host device **308**, which may be a microcontroller, microprocessor, or the like. These portions of code represent Boolean expressions in Conjunctive Normal Form (CNF) and/or Disjunctive Normal Form (DNF) and are off-loaded to the Boolean co-processor to maximize the overall speed of the host device **308**.

**[00175]** Fig. 21 is a high-level block diagram illustrating the functional relationship between a host computer device **308** and a Boolean co-processor **336** having a direct interface with one or more system devices **380**. In this embodiment, greater efficiencies may be achieved by permitting the Boolean co-processor **336** to communicate directly with system devices **380**, rather than gathering device states via the host device **308** and then passing that data on to the Boolean co-processor **336**.

**[00176]** In yet another arrangement, Boolean processing technology may be coupled with a conventional microcontroller, microprocessor, or the like by incorporating the core of a Boolean processor **36**, **136** directly into a host computer device **408**. Fig. 22 is a schematic diagram of a CNF embodiment of a Boolean processor core **436** in accordance with another preferred embodiment of the present invention. In order to simplify the illustration, the CNF Boolean processor core **436** described and illustrated herein utilizes 4-bit addresses, but it will be apparent that this architecture may be scaled to support any address size.

**[00177]** The CNF Boolean processor core **436** includes a next operation address register **442**, and an end of OR address register **444**, an on/off register **437**, a pair of enable registers **439**, a 4-input AND gate circuit **441**, an AND register **454**, an OR conjunct register **458**, an address output AND gate **443** for each of the address registers **442**, **444**, an address OR output circuit **445**, and a plurality of conventional 2-input AND gates **50** and 2-input OR gates **52**. Each of these circuits will be described in more detail hereinbelow.

**[00178]** Fig. 23 is a detailed schematic diagram of a circuit suitable for use as either of the address registers 442, 444 shown in Fig. 22. Like the address registers 42, 44 of the CNF Boolean processor 36 described previously, the two address registers 442, 444 hold addresses used when a particular CNF expression is short circuited. The next operation address is the address of the instruction immediately following the Boolean expression that is being evaluated, and is provided to the next operation address register 442 by the host architecture (not shown) via an input bus. The stored value is then used to short circuit around a CNF expression when any of its conjuncts evaluate to false. Because modifying this address implies the beginning of the evaluation of a CNF Boolean expression, the entire circuit is triggered to an “on” state and the other single bit registers are set/reset to their respective initial states. Similarly, the end of OR address is the address of the instruction immediately following an OR conjunct (CNF conjunct with the terms OR’d together) that is being evaluated, and is provided to the end of OR address register 444 by the host architecture via an input bus. The stored value is then used to short circuit out of an OR conjunct whenever any of its terms evaluate to true. Modifying this address causes the OR conjunct register 458 to be set and an enable bit 439 to be reset. The circuit utilizes exclusive OR gates to provide an indication that the registers value has been modified. This signal is used for setting the on/off register 437 and the OR conjunct register 458 to true, or “on”.

**[00179]** When the various enabling bits are properly set, the CNF Boolean processor core 436 operates to provide a short-circuit function by providing the contents of either the next operation address register 442 or the end of OR address register 444 back to the host architecture as a jump signal to the program counter (not shown), or to any other circuitry that permits the host architecture to jump to an address in its memory and/or microprogram. This function is similar to that of the CNF Boolean processor 36 in that when a particular term in a conjunct evaluates to “true,” then the end of OR address is switched from the end of OR address register 444 through the address OR output circuit 445, and when a particular conjunct evaluates to “false,” then the next operation address is switched from the next operation address register 442 through the address OR output circuit 445. This operation is summarized in Table 5.

Host Condition Bit Status (True/False)	Inside OR Conjunct? (currently evaluating terms OR'd together?)	Jump to Next Conjunct?	Jump Outside of Entire Boolean Expression?
True	No	No	No
True	Yes	Yes	No
False	No	No	Yes
False	Yes	No	No

Table 5 – Summary of Operation of CNF Boolean Processor Core

**[00180]** It should be noted that in an actual implementation of the CNF Boolean processor core 436, two or more of the inputs shown may in fact be combined into single “words” and stored in a combined register. For example, the on/off bit, the condition bit and the reset OR conjunct bit may be combined and set using a single word whose bits correspond to their respective values.

**[00181]** Fig. 24 is a detailed schematic diagram of a circuit suitable for use as any of the 1-bit registers 437, 439, 454, 458 shown in Fig. 22. The circuit is of conventional design and includes a pair of cross-connected NOR gates, forming a simple SR latch. The circuit preferably includes a “set” input and a “reset” input, and the value of the register is provided as an output. However, it will be apparent that the design of any of the register circuits is not limited to SR latches and may comprise any kind of circuit used to implement registers or memory devices. Other than their function, each register 437, 439, 454, 458 may be identical to each other.

**[00182]** Fig. 25 is a detailed schematic diagram of the 4-input AND gate circuit 441 of Fig. 22. The 4-input AND gate circuit 441 includes three AND gates and a NOT gate. One AND gate is used to synchronize a clock or similar signal from the host architecture with an enable bit from one of the enable bit registers 439. When synchronized, the other AND gates are used to propagate the value of a condition bit, received from the host architecture, to either set or reset the AND register 454.

**[00183]** Fig. 26 is a detailed schematic diagram of a circuit suitable for use as either of the address output AND gates 443 shown in Fig. 22. The circuit includes an AND gate

for each addressing bit used by the CNF Boolean processor core 436 and an additional AND gate for gating those address bit AND gates. The two address output AND gates 443 are used to gate the value in the next operation address register 442 and the value in the end of OR address register 444, respectively, to the address outputs of the CNF Boolean processor core 436. The AND gates in the circuit are controlled by the additional AND gate, which propagates the next operation address on the input address bits only when the value of the AND register 454 is false and the value of the on/off register 437 is “true,” or “on.” Likewise, an AND gate propagates the end of OR address on the input address bits only when the value in the OR conjunct register 458 is “true” or one, the condition bit from the host architecture is “true,” or one, and the value of the on/off register 437 is true, or “on.”

**[00184]** Fig. 27 is a detailed schematic diagram of the address OR output circuit 445. The circuit 445 includes a plurality of OR gates, with the number corresponding to the number of addressing bits used by the CNF Boolean processor core 436. The OR gates simply output address data received from the two address output AND gates 443. Because those gates are controlled such that no more than one address output AND gate 443 produces a non-zero address at any given time, the circuit 445 functions simply as a means for funneling data from more than one source to a single output or set of outputs. Thus, it will be apparent that this same function may be provided in other ways, including the replacement of the entire circuit 445 with simple wire cross-connections. In any case, when a short-circuit occurs, either the next operation address or the end of OR address is outputted by the CNF Boolean processor core 436 to the host architecture.

**[00185]** In operation, when the CNF Boolean processor core 436 receives a new next operation address from the host architecture, it turns the processor 436 on by setting the on/off bit to “true,” or “on.” The CNF Boolean processor core 436 then monitors one or more condition bits and evaluates them in conjunction with the status of the AND bit 454, the OR conjunct bit 458, and the enable bits 439 to determine the status of the evaluation. It is these bits that determine the operation of the CNF Boolean processor core 436 and the address result that is subsequently provided back to the host architecture. Notably, although only a single condition bit is illustrated, it should be apparent that multiple condition bits may be incorporated, and that

the bits may optionally be stored in a register in the CNF Boolean processor core **436**. The various condition bits represent the outcome of various evaluations, such as whether the value of a particular device state is equal to, greater than, less than, greater than or equal to, or less than or equal to another value. For the sake of simplicity, however, only a single condition bit is shown.

**[00186]** In addition to receiving condition bit(s), end of OR addresses, next operation addresses, and host clock signals/PC change indicators as input, the CNF Boolean processor core **436** may also receive a reset OR conjunct bit signal and a reset on/off bit signal. The former may be used to signal to the CNF Boolean processor core **436** that the evaluation of an OR conjunct has just completed and prevents the circuit from outputting the contents of the end of OR address register **444** when an OR conjunct is not being evaluated. The latter may be used to reset the on/off register **437** to a value of “false”, or “off.” If the status of this bit is “off,” then the CNF Boolean processor core **436** is prevented from reacting to changes in the condition bit(s), described below, that may occur when the host architecture is processing code that is not a Boolean expression.

**[00187]** A host architecture may likewise utilize a DNF embodiment of a Boolean processor core **536**. Fig. 28 is a schematic diagram of a DNF embodiment of a Boolean processor core **536** in accordance with another preferred embodiment of the present invention. In order to simplify the illustration, the DNF Boolean processor core **536** described and illustrated herein utilizes 4-bit addresses, but it will be apparent that this architecture may be scaled to support any address size.

**[00188]** The DNF Boolean processor core **536** includes an end of operation address register **442**, and an end of AND address register **544**, an on/off register **437**, a pair of enable registers **439**, a 3-input AND gate circuit **541**, an OR register **554**, an AND disjunct register **558**, an address output AND gate **443** for each of the address registers **442**, **544**, an address OR output circuit **445**, a conventional NOT gate **51** and a plurality of conventional 2-input AND gates **50** and 2-input OR gates **52**. Each of these circuits will be described in more detail hereinbelow.

**[00189]** The address registers **442**, **544** are circuits that may be of the type shown in Fig. 23. Like the address registers **42**, **44** of the DNF Boolean processor **136** described previously, the two address registers **442**, **544** hold addresses used when a particular DNF expression is short circuited. The end of operation address is the address of the instruction that is executed when the overall result of the Boolean expression being evaluated is true, and is provided to the end of operation register **442** by the host architecture (not shown) via an input bus. The stored value is then used to short circuit around a DNF expression when any of its disjuncts evaluate to true. Because modifying this address implies the beginning of the evaluation of a DNF Boolean expression, the entire circuit is triggered to an “on” state and the other single bit registers are set/reset to their respective initial states. Similarly, the end of AND address is the address of the instruction immediately following an AND disjunct (DNF disjunct with terms AND’d together) that is being evaluated, and is provided to the end of operation register **544** by the host architecture via an input bus. The stored value is then used to short circuit out of a DNF disjunct when any of its terms evaluate to false. Modifying this address causes the AND disjunct register **558** to be set.

**[00190]** When the various enabling bits are properly set, the DNF Boolean processor core **536** operates to provide a short-circuit function by providing the contents of either the end of operation address register **542** or the end of AND address register **544** back to the host architecture as a jump signal to the program counter (not shown), or to any other circuitry that permits the host architecture to jump to an address in its memory and/or microprogram. This function is similar to that of the DNF Boolean processor **136** in that when a particular term in a disjunct evaluates to “false,” then the end of AND address is switched from the end of AND address register **544** through the address OR output circuit **445**, and when a particular disjunct evaluates to “true,” then the end of operation address is switched from the end of operation address register **442** through the address OR output circuit **445**. This operation is summarized in Table 6.

Host Condition Bit Status (True/False)	Inside AND Disjunct? (currently evaluating terms AND'd together?)	Jump to Next Disjunct?	Jump Outside of Entire Boolean Expression?
True	No	No	Yes
True	Yes	No	No
False	No	No	No
False	Yes	Yes	No

Table 6 – Summary of Operation of DNF Boolean Processor Core

**[00191]** It should be noted that in an actual implementation of the DNF Boolean processor core 536, two or more of the inputs shown may in fact be combined into single “words” and stored in a combined register. For example, the on/off bit, the condition bit and the reset AND disjunct bit may be combined and set using a single word whose bits correspond to their respective values.

**[00192]** The various 1-bit registers 437, 439, 554, 558 are circuits that may be of the type shown in Fig. 24. As described previously, the circuit is of conventional design and includes a pair of cross-connected NOR gates, forming a simple SR latch. The circuit preferably includes a “set” input and a “reset” input, and the value of the register is provided as an output. However, it will be apparent that the design of any of the register circuits is not limited to SR latches and may comprise any kind of circuit used to implement registers or memory devices. Other than their function, each register 437, 439, 554, 558 may be identical to each other.

**[00193]** Fig. 29 is a detailed schematic diagram of the 3-input AND gate circuit 541 of Fig. 28. The 3-input AND gate circuit 541 includes two AND gates. One AND gate is used to synchronize a clock or similar signal from the host architecture with an enable bit from one of the enable bit registers 439. When synchronized, the other AND gate is used to propagate the value of a condition bit, received from the host architecture, to the OR register 554.

**[00194]** The two address output AND gates 443 are circuits that may be of the type shown in Fig. 26. The circuit includes an AND gate for each addressing bit used by the DNF Boolean processor core 536 and an additional AND gate for gating those address bit AND gates. The two address output AND gates 443 are used to gate the end of operation address and the end of AND address, respectively, to the address outputs of the DNF Boolean processor core 536.

The AND gates in the circuit are controlled by the additional AND gate, which propagates the end of operation address on the input address bits only when the value of the OR register **554** is true and the value of the on/off register **437** is “true,” or “on.” Likewise, an AND gate propagates the end of AND address on the input address bits only when the value in the AND disjunct register **558** is “true” or one, the condition bit from the host architecture is “false,” or zero, and the on/off bit is “on.”

**[00195]** The address OR output circuit **445** may be of the type shown in Fig. 27. The circuit **445** includes a plurality of OR gates, with the number corresponding to the number of addressing bits used by the DNF Boolean processor core **536**. The OR gates simply output address data received from the two address output AND gates **443**. Because those gates are controlled such that no more than one address output AND gate **443** produces a non-zero address at any given time, the circuit **445** functions simply as a means for funneling data from more than one source to a single output or set of outputs. Thus, it will be apparent that this same function may be provided in other ways, including the replacement of the entire circuit **445** with simple wire cross-connections. In any case, when a short-circuit occurs, either the end of operation address or the end of OR address is outputted by the DNF Boolean processor core **536** to the host architecture.

**[00196]** In operation, when the DNF Boolean processor core **536** receives a new end of operation address from the host architecture, it turns the processor **536** on by setting the on/off bit to “true,” or “on.” The DNF Boolean processor core **536** then monitors one or more condition bits and evaluates them in conjunction with the status of the OR bit **554**, the AND disjunct bit **558**, and the enable bits **439** to determine the status of the evaluation. It is these bits that determine the operation of the DNF Boolean processor core **536** and the address result that is subsequently provided back to the host architecture. Notably, although only a single condition bit is illustrated, it should be apparent that multiple condition bits may be incorporated, and that the bits may optionally be stored in a register in the DNF Boolean processor core **536**. The various condition bits represent the outcome of various evaluations, such as whether the value of a particular device state is equal to, greater than, less than, greater than or equal to, or less than or equal to another value. For the sake of simplicity, however, only a single condition bit is shown.

**[00197]** In addition to receiving condition bit(s), end of AND addresses, end of operation addresses, and host clock signals/PC change indicators as input, the DNF Boolean processor core **536** may also receive a reset AND disjunct bit signal and a reset on/off bit signal. The former may be used to signal to the DNF Boolean processor core **536** that the evaluation of an AND disjunct has just completed and prevents the circuit from outputting the contents of the end of AND address register **544** when an AND disjunct is not being evaluated. The latter may be used to reset the on/off register **437** to a value of “false,” or “off.” If the status of this bit is “off,” then the DNF Boolean processor core **536** is prevented from reacting to changes in the condition bit(s), described below, that may occur when the host architecture is processing code that is not a Boolean expression.

**[00198]** In still another arrangement, a host architecture may incorporate both a CNF Boolean processor core **436** and a DNF Boolean processor core **536** for added efficiency. Fig. 30 is a schematic diagram of a combined CNF/DNF Boolean processor core **636** in accordance with another preferred embodiment of the present invention. Once again, in order to simplify the illustration, the combination CNF/DNF Boolean processor core **636** described and illustrated herein utilizes 4-bit addresses, but it will be apparent that this architecture may be scaled to support any address size.

**[00199]** The CNF/DNF Boolean processor core **636** includes a combined next operation/end of operation address register **642**, and an end of OR/AND address register **644**, an on/off register **437**, a pair of enable registers **439**, a CNF/DNF register **647**, a 4-input AND gate circuit **441**, an AND/OR register **654**, a combination OR conjunct/AND disjunct register **658**, an address output AND gate **443** for each of the address registers **642**, **644**, an address OR output circuit **445**, a conventional NOT gate **51** and a plurality of conventional 2-input AND gates **50** and 2-input OR gates **52**. Each of these circuits will be described in more detail hereinbelow.

**[00200]** The address registers **642**, **644** are circuits that may be of the type shown in Fig. 23. Like the address registers **42**, **44** of the CNF and DNF Boolean processors **36**, **136** described previously, the two address registers **642**, **644** hold addresses used when a particular CNF or DNF expression is short circuited. In CNF mode, the next operation/end of operation

address is the address of the instruction immediately following the Boolean expression that is being evaluated, while in DNF mode, the next operation/end of operation address is the address of the instruction that is executed when the overall result of the Boolean expression being evaluated is true. In either case, the next operation/end of operation address is provided to the next operation/end of operation address register **642** by the host architecture (not shown) via an input bus. The stored value is then used to short circuit around a CNF or DNF expression when any of its conjuncts or disjuncts, respectively, evaluate to false or true, respectively. Because modifying this address implies the beginning of the evaluation of a new Boolean expression, the entire circuit is triggered to an “on” state and the other single bit registers are set/reset to their respective initial states.

**[00201]**      Similarly, the end of OR/AND address is the address of the instruction immediately following an OR conjunct (CNF conjunct with terms OR’d together) or an AND disjunct (DNF disjunct with terms AND’d together) that is being evaluated and is provided to the next operation/end of operation address register **644** by the host architecture via an input bus. The stored value is then used to short circuit out of a CNF conjunct when any of its terms evaluate to true, or out of a DNF disjunct when any of its terms evaluate to false. Modifying this address causes the OR conjunct/AND disjunct register **658** to be set.

**[00202]**      When the various enabling bits are properly set, the CNF/DNF Boolean processor core **636** operates to provide a short-circuit function by providing the contents of either the next operation/end of operation address register **642** or the end of OR/AND address register **644** back to the host architecture as a jump signal to the program counter (not shown), or to any other circuitry that permits the host architecture to jump to an address in its memory and/or microprogram.

**[00203]**      The various 1-bit registers **437, 439, 647, 654, 658** are circuits that may be of the type shown in Fig. 24. As described previously, the circuit is of conventional design and includes a pair of cross-connected NOR gates, forming a simple SR latch. The circuit preferably includes a “set” input and a “reset” input, and the value of the register is provided as an output. However, it will be apparent that the design of any of the register circuits is not limited to SR

latches and may comprise any kind of circuit used to implement registers or memory devices. Other than their function, each register **437**, **439**, **654**, **658** may be identical to each other.

**[00204]** The 4-input AND gate circuit **441** may be of the type shown in Fig. 25. As described previously, the 4-input AND gate circuit **441** includes three AND gates and a NOT gate. One AND gate is used to synchronize a clock or similar signal from the host architecture with an enable bit from one of the enable bit registers **439**. When synchronized, the other AND gates are used to propagate the value of a condition bit, received from the host architecture, to either set or reset the AND/OR register **654**.

**[00205]** The two address output AND gates **443** are circuits that may be of the type shown in Fig. 26. The circuit includes an AND gate for each addressing bit used by the CNF/DNF Boolean processor core **636** and an additional AND gate for gating those address bit AND gates. The two address output AND gates **443** are used to gate the next operation (or end of operation) address and the end of OR/AND address, respectively, to the address outputs of the CNF/DNF Boolean processor core **636**. The AND gates in the circuit are controlled by the additional AND gate, which propagates the next operation or end of operation address on the input address bits only in the following two scenarios:

- 1) The CNF/DNF register **647** is set for CNF, the AND/OR bit is “false,” and the status of the on/off bit register **437** is “on.”
- 2) The CNF/DNF register **647** is set for DNF, the AND/OR register is “true,” and the status of the on/off bit register **437** is “on.”

**[00206]** Likewise, the end of OR/AND address is propagated on the input address bits in the following two scenarios:

- 1) The CNF/DNF register **647** is set for CNF, the OR conjunct/AND disjunct register **658** is set to “true,” or one, the condition bit from the host architecture is “true,” or one, and the on/off bit **437** is “on.”

2) The CNF/DNF register **647** is set for DNF, the OR conjunct/AND disjunct register **658** is set to “true,” or one, the condition bit from the host architecture is “false,” or one, and the on/off bit **437** is “on.”

**[00207]** The address OR output circuit **445** may be of the type shown in Fig. 27.

The circuit **445** includes a plurality of OR gates, with the number corresponding to the number of addressing bits used by the CNF/DNF Boolean processor core **636**. The OR gates simply output address data received from the two address output AND gates **443**. Because those gates are controlled such that no more than one address output AND gate **443** produces a non-zero address at any given time, the circuit **445** functions simply as a means for funneling data from more than one source to a single output or set of outputs. Thus, it will be apparent that this same function may be provided in other ways, including the replacement of the entire circuit **445** with simple wire cross-connections. In any case, when a short-circuit occurs, either the next operation address or the end of OR/AND address are outputted by the CNF/DNF Boolean processor core **636** to the host architecture.

**[00208]** The CNF/DNF Boolean processor core **636** operates as either a CNF Boolean processor core **436** or a DNF Boolean processor core **536**, depending upon the state of one or more special CNF/DNF bits received from the host architecture and stored in the CNF/DNF register **647**. Combinatorial logic is included to control the rest of the circuit appropriately, but otherwise the operation of the CNF/DNF Boolean processor core **636** is similar to that of the CNF Boolean processor core **436** and DNF Boolean processor core **536**. When the CNF/DNF Boolean processor core **636** receives a new next operation address or end of operation address from the host architecture, it turns the processor **636** on by setting the on/off bit to “true,” or “on.” The CNF/DNF Boolean processor core **636** then monitors one or more condition bits and evaluates them in conjunction with the status of the AND/OR register **654**, the OR conjunct/AND disjunct register **658**, and the enable registers **439** to determine the status of the evaluation. It is these bits that determine the operation of the CNF/DNF Boolean processor core **636** and the address result that is subsequently provided back to the host architecture. Once again, it should be noted that although only a single condition bit is illustrated, it will be apparent that multiple condition bits may be incorporated, and that the bits may optionally be stored in a

register in the CNF/DNF Boolean processor core **636**. The various condition bits represent the outcome of various evaluations, such as whether the value of a particular device state is equal to, greater than, less than, greater than or equal to, or less than or equal to another value. For the sake of simplicity, however, only a single condition bit is shown.

**[00209]** In addition to receiving condition bit(s), end of OR/AND addresses, next operation/end of operation addresses, and host clock signals/PC change indicators as input, the CNF/DNF Boolean processor core **636** may also receive a reset OR conjunct/AND disjunct bit signal and a reset on/off bit signal. The former may be used to signal to the CNF/DNF Boolean processor core **636** that the evaluation of an OR conjunct or AND disjunct has just completed and prevents the circuit from outputting the contents of the end of OR/AND address register **644** when an OR conjunct or AND disjunct, respectively, is not being evaluated. The latter may be used to reset the on/off register **437** to a value of “false,” or “off.” If the status of this bit is “off,” then the CNF/DNF Boolean processor core **636** is prevented from reacting to changes in the condition bit(s), described below, that may occur when the host architecture is processing code that is not a Boolean expression.

**[00210]** In addition, in order to take advantage of the CNF, DNF or CNF/DNF Boolean processor cores **436**, **536**, **636**, conventional compilers would need to be modified slightly. Applications that can reap the benefits of the Boolean processor technology will only need to be re-compiled. The modifications to existing compilers should be minimal. The only changes that need to occur are changes in the way the compilers handle Boolean expressions. The compiler modifications should be designed to assemble the Boolean expressions such that terms that are most likely to trigger short circuiting are evaluated as early as possible in the execution. It should also be designed to group similar types of comparisons (**=**, **!=**, **<**, **>**, etc.) together. In addition, the intelligent compiling process **1000** described previously may be used to re-compile the code containing the Boolean expressions on an ongoing basis.

**[00211]** The CNF/DNF Boolean processor core **636** also requires a small amount of additional overhead since an extra instruction needs to be executed to set the type of Boolean expression being evaluated. This overhead is only incurred when the host architecture with

which the CNF/DNF Boolean processor core **636** is used employs only a single condition bit. In the event that the CNF/DNF Boolean processor core **636** is used with a multiple condition bit host architecture, the CNF/DNF bit can be set in combination with the condition bit set-up circuitry, thus requiring only a single register load operation.

**[00212]** Other variations of a CNF, DNF or CNF/DNF Boolean processor core **436, 536, 636** are also possible. For example, it may not be necessary to include the address registers **442, 444, 544, 644** in the core itself. Instead, any of the Boolean processor cores **436, 536, 636** may take advantage of appropriate registers in the host architecture. Alternatively, the address registers **442, 444, 544, 644** may be replaced by a separate register (not shown), either within the host architecture or within the Boolean processor core **436, 536, 636**, that can be set with a single instruction or a series of instructions and that subsequently sets or resets the values of the appropriate 1-bit registers in the core. Still further, the various 1-bit registers in the core may be set or reset directly with load instructions or any other register-modifying instruction from either the host architecture and/or the Boolean processor core **436, 536, 636**.

**[00213]** The foregoing discussion of Boolean processor cores **436, 536, 636** generally assumes that the output of the core **436, 536, 636** is a direct update of the host architecture's program counter to the instruction address specified in either the next operation address register **442**, end of operation address register **542** or next operation/end of operation address register **642** (as appropriate), or the end of OR register **444**, the end of AND register **544**, or the end of OR/AND register **644** (as appropriate). Alternatively, however, the output of the Boolean processor core **436, 536, 636** may result in the execution of an instruction by the host architecture that makes the value of any of the registers of the Boolean processor core **436, 536, 636** accessible to the host architecture. In yet another alternative, the output of the Boolean processor core **436, 536, 636** may be a feed to an interrupt in the host architecture triggered by the changing of any of the registers of the Boolean processor core **436, 536, 636**. Other outputs or outcomes from the Boolean processor core **436, 536, 636** will also be apparent to those of ordinary skill in the art.

**[00214]** Inclusion of a Boolean processor core **436, 536, 636** in a host architecture has many advantages. Overall, use of a Boolean processor core **436, 536, 636** enables its host architecture to realize a savings in the number of instructions required to evaluate complex Boolean expressions in CNF, DNF, or both. This savings is achieved by adding short circuit capabilities into the hardware of the host architecture, thereby eliminating the need for the branch statements that are normally required to perform short circuiting. As a result, a host architecture with a Boolean processor core **436, 536, 636** can perform up to twice the number of calculations than without the Boolean processor core **436, 536, 636**. Further, with a very small electrical footprint (for example, a CNF Boolean processor core **436** may require only 262 gates to support a 16-bit host architecture), a Boolean processor core **436, 536, 636** can be easily incorporated into a host architecture. In addition, because each Boolean processor core **436, 536, 636** utilizes the instruction set of its host architecture and does not require the addition of new instructions, each is capable of providing backward compatibility with all existing applications. Existing programs will simply ignore the presence of a core **436, 536, 636** unless they are recompiled to take advantage of the enhanced processing benefits.

**[00215]** An exemplary application for the Boolean processor **36** (Figs. 3 and 4) is to manage the state of a set of devices, where the state of one or more devices may necessitate a change in one or more other devices. An example is a home alarm system: “If a door or window is opened while the system is armed, sound an alarm”. The architecture provides a mechanism for a device to report information about itself, that is, its state information, and also a mechanism for changing the state of any device attached to the system. For example, a home automation and alarm system could constantly monitor the state of any number of devices in a house; from doors and windows, to lamps, to televisions, to VCRs. It provides a means for checking the open/closed status of doors and windows, checking whether or not an electrical device is on or off, and also changing the state of the devices (e.g., change the TV channel or turn a lamp on).

**[00216]** Another exemplary use for the Boolean processor **36** is for automobile automation. For example, a proximity sensor could be attached to a car. It is responsible for sensing how close the car is to an object. If the distance between the car and the object closes to within a predetermined distance, the proximity sensor reports a state of ‘too close’ to the Boolean

processor 36. The Boolean processor 36 recognizes this state and initiates a state change to the brake system, thereby slowing the car until a safe distance is achieved.

**[00217]** As described above, the Boolean processor 36 is designed for monitoring and automation applications ranging from small to large-scale. These applications can range from home automation and alarm systems to aeronautical and automobile control systems. The Boolean processor 36 is capable of monitoring any type of device provided that the device meets the following criteria: it is capable of receiving an n-bit address from the processor 36 (this address is used by both the device and the processor 36 to recognize state reporting and enable state changes); it is capable of recognizing its address and reporting its state in an m-bit word, where m is the word size of the device stage storage unit (RAM) 64; and it is capable of recognizing its address and changing its operating state on demand. While the outbound portion of the communications between the processor 36 and the devices it controls is achieved via a direct connection, the inbound portion is achieved by a complementary architecture that polls devices for their states and loads the states in the RAM 64 of the processor 36. In order to meet the above listed requirements for using the processor 36 in practical applications, two complementary architectures have been designed: a device polling unit and a device interface unit.

**[00218]** Fig. 31 is a schematic diagram of one embodiment of the architecture of the device polling unit 66 of the present invention. The device polling unit 66 functions as the main interface between the peripheral devices and the RAM 64. The device polling unit 66 is responsible for finding new devices, assigning device addresses, polling the devices for their current states, and updating the RAM 64 with those states. The device polling unit 66 includes a maximum device address electrically-erasable programmable read-only memory (EEPROM) 68, which stores the highest address of all known devices on the system. The maximum device address EEPROM 68 has an increment line, which increments its value by one whenever it is asserted. It also has  $n$  output lines, which constantly output its value. Its value is incremented when a new device is added to the system via the assertion of the new device found line. The device polling unit 66 also includes an  $n$ -bit incrementing register 70, which holds an  $n$ -bit number representing the current address of the device being polled. It has a reset line, which sets

the register to zero when asserted. It also has an increment line and  $n$  output lines, which constantly output its value to an AND unit 72 and a current address encoder 74. If a new device is not found on the system, the negation of the new device found line asserts the increment line on the register, enabling it to cycle through and poll all of the attached devices by address. The device polling unit 66 includes two AND units 72. One unit allows the value of the maximum device address EEPROM 68 to be sent to a new address encoder 76 if the new address line has been asserted (i.e. a new device has been detected). The second AND unit 72 is used to reset the incrementing register 70 if it equals the maximum device address. The latter is used to conserve clock cycles. Without it, the register would reset upon overflow. In its absence, however, the design would work with  $n$  devices; it saves  $2^n$ -(# of attached devices) clock cycles each time it polls all of the attached devices. The device polling unit 66 includes two address encoders. These units are designed to take  $n$  bits in parallel and output them serially. One encodes new addresses, while the other encodes the address of the device being polled. The device polling unit 66 further includes a device address and state decoder 78, which accepts serial input in blocks of  $n+m$  bits, representing the device address and the state of the device, respectively, and outputs them in parallel to the RAM 64. It should be noted that the encoder and decoder are not mandatory, and are only required in the case that a serial bus device is used.

**[00219]** The device polling unit 66 operates in a continuous loop after it is started. First, it checks for new devices added to the system. If a new device is found (the new device found line is asserted), the device polling unit 66 assigns a system address to it. If a new device is not present in the system, the  $n$ -bit incrementing register 70 is incremented, the device polling unit 66 polls the device corresponding to the address in the incrementing register 70, and then copies the device's current state into the RAM 64. The loop is then repeated. Once the device polling unit 66 is running, it continues to loop, polling for new devices and retrieving device states.

**[00220]** The device polling unit 66 finds new devices by clocking (asserting) the new device seek line. If a new device exists, the new device found line is asserted, incrementing the maximum device address EEPROM 68 and activating the AND gate 72, which allows the address to pass into the new address encoder 76.

**[00221]** Device polling is achieved via the incrementing register **70**, which constantly outputs its value to the current address encoder **74**. It loops through all of the device addresses. The end of the series of devices is recognized when the current device address reaches the maximum device address. This is determined when the result of the current device address AND's with the maximum device address EEPROM's value, resetting the incrementing register **70** to zero. For each address, the device polling unit **66** asserts the state enable line, requesting the device's state. When a device detects its address on the state enable line, it outputs (e.g., serially) its address and state on the device state line. The device address and state decoder **78** then outputs the  $n+m$  bits (representing the device address and state, respectively) to the RAM **64**.

**[00222]** Fig. 32 is a schematic diagram of one embodiment of the architecture of the device interface unit **80** of the present invention. Each device that interfaces with the Boolean processor **36** (Figs. 3 and 4) is assumed to be a relatively intelligent device that can accept an  $n$ -bit address and has up to  $m$  states. Accordingly, the device interface unit **80** includes a new device EEPROM **82**, which is a 1-bit store that is set to one when the device is built. When the new device seek line is asserted, this bit (on a new device) will assert the new device found line. The device interface unit **80** also includes an address decoder **84**. If the new device EEPROM bit is set, it will allow the address passed on the new address line to be placed in the  $n$ -bit address EEPROM **86** and then clear the new device EEPROM **82**. Once the new device EEPROM **82** has been cleared, it will only pass addresses to the AND gate **72** that tests to see if the device is being asked for its state. The device interface unit **80** further includes a control word decoder **88**, which reads the serial bits off of the control line. If the address matches the address in the address EEPROM, the control bits output to the device controller to change its state. It is reset via internal logic that counts the assertion of the new device seek line and resets every  $n+m$  clock cycles. The device interface unit **80** further includes the  $n$ -bit address EEPROM **86**, which stores the address of the device and constantly outputs it, an  $m$ -bit state register **90**, which holds the current state of the device and has an output enable line that allows it to output its value, and an address/state encoder **92**, which accepts an  $n$ -bit address and an  $m$ -bit state and outputs them (e.g., serially) on the device state line.

**[00223]** The device interface unit **80** is designed to listen for the following assertions: New Device Seek, New Address, State Enable, and Control Line. The unit determines whether or not it has any work to do as a result of any such assertion. If so, it may assert any of the following back to the caller: New Device Found and Current State of the Device. When a device is attached to the bus, its value for the new device EEPROM **82** is set to ‘1’. This indicates that it has not yet been incorporated into the system. When the new device seek line is asserted, its value (‘1’) is passed to an AND gate **72** along with the value (‘1’) for the new device EEPROM **82**. If it is a new device, i.e. the result of the AND is ‘1’, the new device found line is asserted, informing the device polling unit **66** of the existence of a new device.

**[00224]** By default, the device interface unit **80** “listens” for a new address on the new address line. The assertion of the new device found line forces the device polling unit **66** to return the next device address. The new address is placed in the *n*-bit address EEPROM **86**. The address decoder **84** then clears the new device EEPROM **82**. The next time the device receives the new device seek line assertion, it does not assert the new device line. The device has now been assimilated. Once assimilated, the device may be polled for its state. During the polling phase of the device polling unit **66**, each device is queried by its address. When queried, the device interface unit **80** recognizes its address and returns its current state. When the state enable line is asserted to the device, the address decoder **84** compares the address on the line with the device address stored in the address EEPROM **86**. This comparison is performed via an AND gate **72**. If the addresses match, then the request for state information is directed to this device. The positive result of the AND causes the output enable line to the state register to be asserted and the address/state encoder **92** to be enabled. The state information is sent to the address/state encoder **92**. The address/state encoder **92** accepts the *n*-bit address and the *m*-bit state and outputs them serially or in parallel on the device state line(s).

**[00225]** If the Boolean processor **36** detects a combination of states that requires a change in another state, it will send the information over the control line. Each device interface unit accepts and reads the data from the asserted control line. The control word decoder **88** compares the incoming address to the address in the address EEPROM **86**. If the addresses

match, the request to make a state change is made to the current device. The control bits are then output to the device controller to initiate a change to its state.

**[00226]** Fig. 33 is a schematic diagram of one embodiment of a system that includes a CNF Boolean processor 36 and the device polling unit 66 of the present invention, and Fig. 34 is a schematic diagram of one embodiment of a system that includes  $n$  devices, each incorporating the device interface unit 80 of the present invention. Referring to Figs. 33 and 34, one of the advantages of having a system including the Boolean processor 36, the device polling unit 66, and a plurality of devices 94 is that the slowest operation of the system, namely the polling of devices, is decoupled from the processing performed by the Boolean processor 36. This allows the Boolean processor 36 to run at full speed, unencumbered by the relative speed of the device polling unit 66 and the device bus.

**[00227]** Overall reliability and integrity of the data in the device state storage 64 may be enhanced by including additional logic designed to properly synchronize operation of the Boolean processor or processor core with the process of updating the state data in the device state storage 64. To ensure system accuracy, it is important that a memory location is not read while its contents are being modified. Doing so could result in erroneous results. To ensure that the aforementioned situation does not occur, the Boolean Processor architecture may be modified so that the processor either waits for the modification operation to end before reading a location in memory, or skips the operation. In the event that the value of the memory location is critical to the operation being performed by the system, putting the system in a wait state is preferable.

**[00228]** The addition of a wait state or skip operation can be achieved by adding some form of indicator, including, but not limited to, a single bit added to each memory location, that will indicate whether or not a memory location is in the process of being modified. The processor will then wait for the modification to end before accessing the location or skip the reading of the location.

**[00229]** Additionally, the processor may write state change information directly across a bus to devices attached to it. Fig. 35 is a high-level block diagram illustrating a first

approach for ensuring data integrity in a Boolean processor system. The processor may be designed to process its micro-program at a much faster rate than its devices operate. In the event that two or more device states are changed in a timeframe that is shorter than the time required to update a device's state, a buffer 61, as illustrated in Fig. 35, will need to be used to store state change commands until the bus is clear. Once an update message has been sent, the next message in the buffer 61 may be sent. To ensure system accuracy, the single bit in RAM, mentioned above, that indicates that the state of the corresponding device is being modified can be set to true to prevent the system from reading an expired state from device state storage.

**[00230]** Another method for updating device states would be the addition of another RAM module 63 that will store updated states. Fig. 36 is a high-level block diagram illustrating a second approach for ensuring data integrity in a Boolean processor system. The Boolean processor would write state updates to that module 63. The devices could then request their own updated states from the RAM module 63. This configuration would operate similarly to the configuration described above except that instead of having device states "pushed" out to the devices via the control encoder 62, devices would "pull" their states from the additional RAM module, as illustrated in Fig. 36.

**[00231]** An exemplary Boolean processor-based system is a home automation/alarm system. The Boolean processor 36 monitors and controls, for example, 256 devices ( $n = 8$ ), each device having, for example, 256 states ( $m = 8$ ). The system includes, for example, a door, a window, a lamp, and a motion detector. In addition to these units, the system uses a clock, an arm/disarm unit, and a siren. Although the majority of possible device states and control words are not used in this example, the full eight bits for addressing, state reporting, and state changes are used. Each device functions as follows:

**Example 10:**

Device	States/Control Functions	8-bit value
Door Sensor (System Address = 00000001)	Open	00000000
	Closed	00000001

Window Sensor (System Address = 00000010)	Open Closed	00000000 00000001
Motion Detector (System Address = 00000011)	No Motion Motion Occurring	00000000 00000001
Lamp (System Address = 00000100)	Off On Turn Off Turn On Flash Stop Flashing (return to pre-flash state)	00000000 00000001 00000000 00000001 00000010 00000011
Clock (System Address = 00000101)	Get Current Time  Ex: 12:30 am = 00000011 10:10 am = 00111101	Value corresponds to 10 minute increments from 00:00.
Arm/Disarm Unit (System Address = 00000110)	System Disarmed System Armed Disarm System Arm System	00000000 00000001 00000000 00000001
Siren (System Address = 00000111)	Off On	00000000 00000001

**[00232]** In addition to the above assumptions, it is assumed that a personal computer (PC) is interfaced with the system and is used to translate code into micro-code and to load the control store. The home automation/alarm system functions as follows: at 6:00 am, disarm the alarm system; at 8:30 am, arm the alarm system; at 5:00 pm, disarm the alarm system; at 5:30 pm, turn the lamp on; at 10:30 pm, arm the alarm system; and at 10:30 pm, turn the lamp off.

**[00233]** If the alarm system is armed and the door or window is open, the siren sounds and the light flashes until the alarm system is disarmed. The high-level code entered into the PC is as follows:

- 10: if time = 6am then arm/disarm = disarm;
- if time = 8:30am then arm/disarm = arm;
- if time = 5pm then arm/disarm = disarm;

```

if time = 5:30pm then lamp = on;
if time = 10:30pm then arm/disarm = arm;
if time = 10:30pm then lamp = off;
if arm/disarm = armed and (door = open or window = open) then
    while arm/disarm = armed
        siren = on;
        lamp = flash;
    end while;
goto 10;

```

**[00234]** The compiled micro-program for this functionality is illustrated in Table 7.

Control Store Address	Instruction Register			Functionality
	Address	Control/State	Opcode	
00000000	00000011	00000000	110	Start of Boolean expr.
00000001	00000101	00100100	000	Time = 6am? (AND)
00000010	00000110	00000000	010	Disarm system if TRUE.
00000011	00000110	00000000	110	Start of Boolean expr.
00000100	00000101	00110011	000	Time = 8:30am? (AND)
00000101	00000110	00000001	010	Arm system if TRUE
00000110	00001001	00000000	110	Start of Boolean expr.
00000111	00000101	01100110	000	Time = 5pm? (AND)
00001000	00000110	00000000	010	Disarm system if TRUE
00001001	000001100	00000000	110	Start of Boolean expr.
00001010	00000101	01101001	000	Time = 5:30pm? (AND)
00001011	00000100	00000001	010	Turn lamp on if TRUE
00001100	00000111	00000000	110	Start of Boolean expr.
00001101	00000101	10000111	000	Time = 10:30pm? (AND)
00001110	00000110	00000001	010	Arm System if TRUE
00001111	00010010	00000000	110	Start of Boolean expr.
00010001	00000101	10000111	000	Time = 10:30pm? (AND)
00010001	00000100	00000000	010	Turn lamp off if TRUE
00010010	00011001	00000000	110	Start of Boolean expr.
00010011	00000110	00000001	000	System Armed? (AND)
00010100	00010111	00000000	111	Start of Conjunction
00010101	00000001	00000000	001	Door Open? (OR)

00010110	00000010	00000000	001	Window Open? (OR)
00010111	00000111	00000001	010	Turn Siren on if TRUE
00011000	00000100	00000010	010	Flash lamp if TRUE
00011001	00011101	00000000	110	Start of Boolean expr.
00011010	00000110	00000000	000	System Disarmed? (AND)
00011011	00000111	00000000	010	Turn siren off
00011100	00000100	00000011	010	Stop flashing lamp
00011101	00000000	00000000	100	Loop to beginning of control store

Table 7 – Home Automation/Alarm System Micro-program

**[00235]** The performance of a Boolean processor **36, 136** may be further enhanced by the inclusion of an enhanced logic unit **210, 310**. capable of providing such additional functionality as comparing a device state to a threshold value, comparing one device state (or the value from one memory location) to another device state or value from memory, loading a value directly into a particular memory location, or the like, or a combination thereof.

**[00236]** Fig. 37 is a schematic diagram of a CNF Boolean processor **236** having a first enhanced logic unit **210**, and Fig. 38 is an enlarged schematic diagram of a portion of the CNF Boolean processor **236** of Fig. 37. The first enhanced logic unit **210** may incorporate one of the AND gates **50** along with a comparator **211** and a switch **213**. The operation of the switch **213** is controlled by the operation decoder **60** such that the switch **213** passes the output of either the AND gate **50** or the comparator **211** to the OR gate **52**, depending on the control input received from the operation decoder **60**. As with the Boolean processors **36, 136** of the previous embodiments, the AND gate **50** simply indicates whether a device state value (the value from a memory location) is equal to a specific value received from the control/state portion of the instruction register **40**. On the other hand, the comparator **211** is a simple device that compares a device state value to a threshold value, likewise received from the control/state portion of the instruction register **40**, to determine whether the device state value is greater than (or, depending on how the comparator **211** is implemented, less than, greater than or equal to, or less than or equal to) the threshold value. This information is then reflected in the output of the comparator **211**, which is generally a single bit.

**[00237]** In order to utilize this additional functionality, two new instructions, represented by two new operation codes (Op Codes 8 and 9) may be provided. Op Code 8—(AND Compare) enables the AND gate **50** that loads the AND register **54** in the event that the conditional state of the device at the address specified in the instruction register **40** meets the threshold requirement specified by the value in control/state portion of the instruction register **40**. Op Code 9 (OR Compare) sets the value of the OR conjunct register **58** to one, which enables short-circuiting (based on a threshold being met) within a conjunct containing OR clauses. As perhaps most easily understood from Fig. 38, it will be appreciated that the operation of these two instructions is similar to the operation of the instructions represented by Op Codes 0 and 1, respectively (Boolean AND and Boolean OR), except that AND Compare and OR Compare utilize a threshold comparison instead of an examination for exact equality. The comparator **211** and the adjacent AND gate **50** each receive the same input data, from the instruction register **40** and the device state storage **64**, respectively, and their respective outputs are used in the same way that the output of the AND gate **50** is in the other embodiments of the Boolean processor **36**, **136**. The only difference is in which input source (the AND gate **50** or the comparator **211**) is provided as an output to the rest of the processor **236**, as controlled by the operation decoder **60** on the basis of the operation code.

**[00238]** It will also be appreciated that the inclusion of two new Op Codes may require an increase in the size of the operation code portion of the instruction register **40**, such as from 3 bits to 4 bits. Alternatively, however, the two new instructions proposed here may replace Op Codes 0 and 1 respectively, since their operation is so similar, as described previously.

**[00239]** Fig. 39 is a schematic diagram of a CNF Boolean processor **336** having a second enhanced logic unit **310**, and Fig. 40 is an enlarged schematic diagram of a portion of the CNF Boolean processor **336** of Fig. 39. The second enhanced logic unit **310** may incorporate the AND gate **50**, the comparator **211** and the switch **213** of the first enhanced logic unit **210** as well as two additional switches **315**, **317**. Each switch **315**, **317** is arranged to receive input from both device state storage **64** and from the control/state portion of the instruction register **40**. The output of one switch **315** is connected to an input of the AND gate **50**, while the output of the

other switch 317 is connected to an input of the comparator 211. Like the first switch 211, the operation of the additional switches 315, 317 are controlled by the operation decoder 60, such that each switch 315, 317 passes on either the data from the instruction register 40 or the device state storage 64. The output from one of the switches 315 is passed to the AND gate 50, while the output from the other switch 317 is passed to the comparator 211.

**[00240]** Like the first enhanced logic unit 210, the second enhanced logic unit 310 is capable of both determining whether two input values are equal to each other and determining whether one input value meets a threshold requirement established by a second input value. In the first enhanced logic unit 210, however, only one of the input values is a device state from device state storage 64, while the second input value is received directly from the control/state portion of the instruction register 40. In the second enhanced logic unit 310, on the other hand, one device state from the device state storage 64 may be compared to a second device state using either the AND gate 50 or the comparator 211. Finally, as with the first enhanced logic unit 210, the original switch 213 is controlled by the operation decoder 60 such that the switch 213 passes the output of either the AND gate 50 or the comparator 211 to the OR gate 52, depending on the control input received from the operation decoder 60. This is perhaps most easily understood with reference to Fig. 40.

**[00241]** The second enhanced logic unit 310 also includes an additional feature. As shown in Figs. 39 and 40, a direct connection 319 exists between the control/state portion of the instruction register 40 and the device state storage 64. This connection permits a value to be loaded directly from the control/state portion of the instruction register 40 to the device state storage 64.

**[00242]** Figs. 41 and 42 are enlarged schematic diagrams of portions of two different Boolean processors 736, 836 having a direct connection 319 for facilitating inter-device communication via the device state storage 64. Each processor 736, 836 used a direct connection 319, similar to that shown in Figs. 39 and 40, to provide the state of any device to any other device in the system. As shown in Fig. 41, an extra instruction may be provided to facilitate this transfer by signaling the device state storage 64 or combinatorial logic (not shown) associated therewith. Alternatively, as shown in Fig. 42, an extra bit 35 may be added to the

instruction register **40** in order to indicate that the value in the control/state portion of the instruction register **40** is either a direct value or the address of another device. In the latter case, the state of the memory location corresponding to the source device would be sent to the memory location corresponding to the target device.

**[00243]** In order to utilize all of this additional functionality, five new instructions, represented by five new operation codes (Op Codes 10, 11, 12, 13, 14 and 15) may be provided. Op Code 10—(AND Compare Memory to Memory) and Op Code 11—(OR Compare Memory to Memory) are similar to Op Code 8 (renamed AND Compare w/Immediate Value) and Op Code 9 (renamed OR Compare w/Immediate Value), respectively, except that the comparisons are between two values from device state storage **64**, rather than between one value from device state storage **64** and one value from the control/state portion of the instruction register **40**. Op Code 12—(Boolean AND, Memory to Memory) and Op Code 13—(Boolean OR, Memory to Memory) are similar to Op Code 0 and Op Code 1, respectively, except that the tests for equality are made between two values from device state storage **64**, rather than between one value from device state storage **64** and one value from the control/state portion of the instruction register **40**. Op Code 14—(Load Memory w/Immediate Value) loads the current value in the control/state portion of the instruction register **40** into the memory location indicated by the current value of the address portion of the instruction register **40**. This latter instruction, coupled with the physical connection described previously, simplifies the process of loading a value into device state storage **64**. Without this instruction, loading a value into device state storage **64** would require the evaluation of a simple CNF expression in which the outcome is true, followed by an end of operation instruction (Op Code 2) that would update the location in device state storage **64**. Finally, Op Code 15—(Load Memory w/Device State) accepts the addresses of two devices (between which state information is to be transferred) and facilitates the exchange of state data as shown in Fig. 41.

**[00244]** It will be appreciated that the inclusion of five new Op Codes may not require an increase in the size of the operation code portion of the instruction register **40**, relative to the size required by the first enhanced logic unit **210**, if the operation code portion already requires 4 bits, since 4 bits will support up to a total of **16** operation codes. However, although

not illustrated herein, it will also be appreciated that the Boolean OR/Memory to Memory and Boolean AND/Memory to Memory instructions (Op Code 12 and Op Code 13, respectively) may be utilized even without the presence of the comparator 211 and the four “Compare” instructions (Op Codes 8, 9, 10 and 11), and without the presence of the direct connection between the control/state portion of the instruction register 40 and the device state storage 64 and the “Load Memory” instruction (Op Code 14). Finally, although likewise not illustrated herein, it will further be appreciated that the “Compare” instructions, the “Memory to Memory” instructions, or both may be utilized in a DNF Boolean processor 136 in the same manner as with the CNF processor 36 illustrated in Figs. 37-40.

**[00245]** With regard to the present invention, it is apparent that there has been provided a Boolean processor. The architecture of the Boolean processor is optimized for monitoring and automation applications. The relatively small instruction set and design of the Boolean processor provide an instruction savings of up to about 87.5% in relation to typical microprocessor and microcontroller instruction sets. These instruction savings and simple design provide the Boolean processor with high speed, in terms of instructions, as compared to other general-purpose architectures performing similar functions. In addition to efficiency, the architecture of the Boolean processor is scalable. For example, if the Boolean processor is built with 32-bit addresses and 32-bit states, it can handle over about 4 billion ( $2^{32}$ ) devices, each with over about 4 billion possible states. The speed and scalability of the architecture of the Boolean processor make it a good candidate for large, critical applications, such as aeronautical and automotive monitoring, control, and automation applications.

**[00246]** As the number of sensors, or devices, increases, so does the amount of wiring required for communications. Thus, serial communications may be used with the Boolean processor. Another advantage of the architecture of the Boolean processor is that it may be fitted with either a parallel or serial communications bus.

**[00247]** Multiple Boolean processors may also be employed for greater efficiencies. For example, multiple Boolean processors 36, 136 or processor cores 436, 536, 636 may be used in parallel to evaluate complex CNF or DNF expressions in a divide-and-conquer type mode. In the case of CNF, the expression’s conjuncts would be distributed to the different

processors for evaluation. In the event that a conjunct, or series of conjuncts, resulted in a false evaluation, the processor upon which the conjunct(s) were being evaluated would signal the other processors that the entire operation was false, thereby causing the next Boolean expression to be distributed among the processors for evaluation. Similarly, in the case of DNF, the expression's disjuncts would be distributed to the different processors for evaluation. In the event that a disjunct, or series of disjuncts, resulted in a true evaluation, the processor upon which the disjunct(s) were being evaluated would signal the other processors that the entire operation was true, thereby causing the operation that executes upon a true result to be executed as well as the next expression to be distributed among the processors for evaluation.

**[00248]** Instances of both CNF and DNF Boolean processors may also be intermingled to handle expressions. For example, Fig. 43 is a high-level block diagram illustrating the use of Boolean processors **36**, **136** in a parallel processing environment. In this illustration, two DNF Boolean processors **136** and one CNF Boolean processor **36** may be utilized to the following CNF expression:

*If (A or B or C or D) and (E or F or G) and H then I*

where A, B, C, D, E, F, G, and H are terms of the form  $x = y$  and where x represents a device state and y represents a value for comparison. The two DNF Boolean processors **136** could be employed to evaluate the first two conjuncts since each conjunct represents a DNF expression in its most simple form (i.e., comprised entirely of single term disjuncts). The final values from the DNF Boolean processors as well as the evaluation of H could then be rolled into a CNF Boolean processor **36** as shown in Fig. 43.

**[00249]** A plurality of Boolean processors may also be used in conjunction with different but related systems, each employing a Boolean processor designed to handle a large number of sensors or devices specific to the given system. The individual systems can communicate via another, smaller Boolean processor that is linked to each of the systems as one of their devices. The smaller Boolean processor handles interactions among the systems. For example, consider a braking system and a speedometer system in an automobile. The braking system can be outfitted with numerous devices and sensors to control the application of the brakes, monitor temperature, and monitor pad wear, to name a few. Other systems in the car

may only need to know whether or not the brakes are being applied and whether or not there is a problem with the entire braking system. The speedometer system can also be outfitted with numerous devices and sensors for monitoring its own health. Like the braking system, it only needs to communicate speed and generic warnings to the other systems in the car. Because each device only needs to communicate two states, a smaller Boolean processor with a smaller bus that controls the interaction between these systems can be used, thereby saving wiring weight and confining complex communications infrastructure to small areas of the vehicle.

**[00250]** Another potential use for the Boolean processor is as an interrupt controller. A Boolean processor-based controller can enable a microprocessor to be interrupted by an almost limitless number of devices. The Boolean processor acts as an “interrupt broker” for the devices attached to it.

**[00251]** Although the Boolean processor of the present invention has been described and illustrated with reference to preferred embodiments and examples thereof, other embodiments and examples may be used and the following claims are intended to cover all such equivalents.

**[00252]** Based on the foregoing information, it is readily understood by those persons skilled in the art that the present invention is susceptible of broad utility and application. Many embodiments and adaptations of the present invention other than those specifically described herein, as well as many variations, modifications, and equivalent arrangements, will be apparent from or reasonably suggested by the present invention and the foregoing descriptions thereof, without departing from the substance or scope of the present invention. Accordingly, while the present invention has been described herein in detail in relation to its preferred embodiment, it is to be understood that this disclosure is only illustrative and exemplary of the present invention and is made merely for the purpose of providing a full and enabling disclosure of the invention. The foregoing disclosure is not intended to be construed to limit the present invention or otherwise exclude any such other embodiments, adaptations, variations, modifications or equivalent arrangements; the present invention being limited only by the claims appended hereto and the equivalents thereof. Although specific terms are employed herein, they are used in a generic and descriptive sense only and not for the purpose of limitation.